PhD-FSTC-2019-01
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defence held on 11/01/2019 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

INES HAJRI
Born on 13th May 1986 in Comores (Comores)

SUPPORTING CHANGE IN PRODUCT LINES WITHIN THE CONTEXT OF USE CASE-DRIVEN DEVELOPMENT AND TESTING

DISSERTATION DEFENSE COMMITTEE

PROF. DR. LIONEL C. BRIAND, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

PROF. DR. ALEXANDER EGYED, MEMBER
Professor, Johannes Kepler University, Austria

DR. SHIVA NEJATI, Chairman
Senior Research Scientist, University of Luxembourg, Luxembourg

DR. STEFANIA GNESI, MEMBER
Chief Scientist, Alessandro Faedo Institute of Information Science & Technology, Italy

DR. FABRIZIO PASTORE, Vice Chairman
Research Scientist, University of Luxembourg, Luxembourg

Abstract

Product Line Engineering (PLE) is a crucial practice in many software development environments where systems are complex and developed for multiple customers with varying needs. At the same time, many business contexts are use case-driven where use cases are the main artifacts driving requirements elicitation and many other development activities. This is also the case for IEE S.A., a leading supplier of embedded systems in the automotive domain. This dissertation is motivated by the discussions with IEE which aims to adopt PLE in its software development practice. The current development practice at IEE is use case-driven and based on *clone-and-own reuse*. IEE starts a new product with an initial customer. IEE analysts elicit requirements as use case and domain models. Then, they derive system test cases from the use case models for the initial customer. For each new customer of the product, IEE analysts need to clone the current models, and negotiate variabilities with the customer to produce new use case and domain models, and to derive and select new system test cases from the updated use cases (i.e., change management for use cases and regression test selection for system test cases). With such practice, IEE analysts lose track of commonalities and variabilities across products. They, together with the customer, need to evaluate the entire use cases, domain model and system test cases.

The *clone-and-own reuse* practice is fully manual, error-prone and time-consuming in industrial settings, which leads to ad-hoc change management for use cases, domain models and test cases in the context of product lines since the variability information is not explicitly represented.

The need for supporting PLE in the context of use case-driven development and testing has been already acknowledged in the literature. Most of the existing approaches rely on feature modeling, including establishing and maintaining traces between feature models and use case models and between feature models and system test cases. Due to limited resources, many software development companies find such additional traceability and maintainability effort to be impractical. In addition, most of the approaches do not provide automated support for requirements evolution in a product family in terms of change impact analysis and regression testing. In this dissertation we address the following problems which arise in managing changes in use case models and system test cases in a product family.

Modeling Variability in Requirements with Additional Traceability to Feature Models. The analysts need to explicitly document variability information (e.g., variant use cases, variation points, and optional steps) for use cases and a domain model, which is the basis on which to configure products with customers. Relating feature models to use cases and domain model is the most straightforward option but has shortcomings in terms of additional modeling and traceability effort. For example, it is not easy for analysts and customers to comprehend and visualize all variability information traced to use case diagrams, use case specifications and domain models. In our industrial case study, we identified 15 mandatory and 14 variant use cases which contain 8 variation points, and 7 variant dependencies. The use cases include 244 use case flows (29 basic flows, 202 alternative flows, and 13 optional alternative flows) with 27 optional steps while the

domain model contains 11 variant domain entities. The variability information scattered across all these use case flows with trace links from feature models would need to be communicated to customers and used to configure a product.

Manual, Expensive and Error Prone Configuration of Product Specific Requirements. In order to facilitate use case-driven configuration in industrial practice, a high degree of automation is a must while the analysts are interactively guided for their configuration decisions. Before the configuration, it should be automatically confirmed that all artifacts with variability information, including use case diagram, specifications, and domain model, are consistent. Any inconsistency in these artifacts may cause invalid configuration outputs. Adding to the complexity affecting the decision-making process during configuration, there may be contradicting decisions and hierarchies among decisions. Changes on configuration decisions may have impact on prior decisions as well as on subsequent decisions. During the configuration process, the analysts need to be interactively informed about contradicting decisions, the order of possible decisions, and the impact of decision changes on other decisions. Without interactive guidance and proper tool support, the analysts have to manually identify and fix inconsistent product line artifacts, resolve decision contradictions, and change subsequent decisions, which leads to the time-consuming, expensive and error prone configuration of use case and domain models for a product.

Manual and Expensive Regression Testing in Product Families. The current use case-driven testing practice in many software development environments follows the testing strategy referred to as *opportunistic reuse of test assets* for product families. When there is an initial customer for a product in the product family, the product requirements are elicited from the initial customer and documented as a use case diagram and use case specifications. System test cases are then generated from the use cases for the initial customer. For each new customer in the product family, test engineers manually choose and prioritize, from the existing test suite(s) for the initial/previous product(s), test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly in the new product. This form of test reuse is not performed systematically, which means that there is no structured, automated method that supports the activity of selecting and prioritizing test cases. The current practice is fully manual, error-prone and time-consuming, which leads to ad-hoc change management for system test cases in product lines. Therefore, product line modeling and testing techniques are needed to automate the reuse of system test cases in the context of use case-driven development of a product family.

In this dissertation we provide a configuration framework for use case models and system test cases in product families. We choose PLE as a solution platform for our approach. PLE provides a way to engineer a set of related products as a product family in an efficient manner, taking advantage of the products' similarities (commonalities) while managing their differences (variabilities). It enables the analysts to configure products in a product family by making configuration decisions about the product variabilities. To give an explicit structure of use cases and their trace links to system test cases, we use metamodels and models within the context of Model Driven Engineering (MDE). The dissertation provides the following contributions:

A modeling method for capturing variability information in Product Line (PL) use case and domain models. The modeling method enables analysts to capture and document variability in PL use case diagrams, use case specifications, and domain model, without further requiring any feature model. For PL use case diagrams, it uses some extensions in the literature which overcome the shortcomings of textual representations of variability, such as implicit variants and variation points. Further, for PL use case specifications, we employ Restricted Use Case Modeling (RUCM), which includes a template and restriction rules to reduce imprecision and incompleteness in use cases. RUCM was a clear choice since it reduces ambiguity and facilitates automated analysis of use cases. However, since it was not originally meant to model variability, we introduce some PL extensions to capture variability in use case specifications. To be able to capture variability in PL domain models, we rely on the stereotypes (i.e., *variation*, *variant* and *optional*), already proposed in the literature.

An approach for automated configuration of Product Specific (PS) use case and domain models. The use case-driven configuration approach is built on top of PUM. It provides a degree of configuration automation that enables effective product-line management in use case-driven development, without requiring additional modeling artifacts and traceability effort. Our approach supports four activities. First, the analysts model the variability information explicitly in a PL use case diagram, its use case specifications, and its corresponding domain model. Second, the consistency of the PL use case diagram and specifications are checked and inconsistencies are reported if there are any. For instance, a variation point in the use case diagram might be missing in the corresponding use case specification or a use case specification may not conform to the extended RUCM template. Third, the analyst is guided to make configuration decisions based on variability information in the PL models. The partial order of decisions to be made is automatically identified from the dependencies among variation points and variant use cases. In the case of contradicting configuration decisions, such as two decisions resulting in selecting variant use cases violating some dependency constraints, we automatically detect and report them. The analyst must then backtrack and revise the decisions to resolve these inconsistencies. Fourth, based on configuration decisions, the PS use case and domain models are generated from the PL use case and domain models. To support these activities, we developed a tool, *PUMConf (Product line Use case Model Configurator)*. The tool automatically checks the consistency of the PL models, identifies the partial order of decisions to be made, determines contradicting decisions, and generates PS use case and domain models.

A change impact analysis approach for evolving configuration decisions in PL use case models. The change impact analysis approach, based on our use case-driven modeling and configuration techniques, supports the evolution of configuration decisions. We do not address here evolving PL use case models, which need to be treated in a separate approach. Change impact analysis provides a sound basis to decide whether a change is worth the effort and which decisions should be changed as a consequence. In our context, we aim to automate the identification of decisions impacted by changes in configuration decisions on PL use case models. Our approach supports three activities. First, the analyst proposes a change but does not apply it to the corresponding configuration decision. Second, the impact of the proposed change on other configuration decisions for the PL use case diagram are automatically identified. In the PL use case diagram, variant use cases and variation points are connected to each other with some dependencies, i.e., *require*, *conflict* and *include*. In the case of a changed diagram decision contradicting prior and/or subsequent diagram decisions, such as a subsequent decision resulting in selecting variant use cases violating some dependency constraints because of the new/changed decision, we automatically detect and report them. To this end, we developed an algorithm, which enables reasoning on subsequent decisions as part of our impact analysis approach. The analyst is informed about the change impact on decisions for the PL use case diagram. Based on this, the analyst should decide whether the proposed change is to be applied to the corresponding decision. Third, the PS use case models are incrementally regenerated only for the impacted decisions after the analyst actually makes all the required changes. To do so, we implemented a model differencing pipeline which identifies decision changes to be used in the reconfiguration of PS models. There are two sets of decisions: (i) the set of previously made decisions used to initially generate the PS use case models and (ii) the set of decisions including decisions changed after the initial generation of the PS models. Our approach compares the two sets to determine for which decisions we need to incrementally regenerate the PS models. To support these three activities, we extended our configurator, PUMConf.

An approach for automated classification and prioritization of system test cases in a family of products. The automated classification and prioritization approach is based on our use case-driven modeling and configuration techniques. It supports product line testing for evolving products of a product family in terms of evolving configuration decisions in PL use case models. In our context, we aim to automate the identification of system test cases impacted by changes in configuration decisions in PL use case models when a new product is configured in the product family. The initial product is tested individually and the following products are tested using regression testing techniques, i.e., test case selection and prioritization based on configuration decision changes between the previous product(s) and the new product to be tested. Our approach supports two activities. First, the system test cases of the previous product(s) are automatically classified as *obsolete*, *retestable*, and *reusable*. An *obsolete* test case cannot be executed on the new product as the corresponding use case scenarios are not selected for the new product. A *retestable* test case is still valid but needs to be rerun to determine the possible impact of changes whereas a *reusable* test case is also valid but does not need to be rerun for the new product. We also identify the use cases of the new product that have not been tested so far in the product family. To do so, we reused our model differencing pipeline which identifies decision changes to be used in the reconfiguration of PS models. Our approach classifies the configuration decisions as *new*, *deleted* and *updated* to identify the impacted parts of the use case models of the previous product(s). By using the traces from the impacted parts of the use case models to system test cases, we automatically classify the system test cases for test case selection. Second, the system test cases are automatically prioritized based on multiple risk factors such as fault proneness of requirements and requirements volatility in the product line. To this end, we developed a prediction model that computes a prioritization score for each system test case based on these factors. To support these activities, we extended our tool, PUMConf.

Acknowledgements

To the memory of my father, Hajri Taher, who has been a wonderful dad. May your soul rest in peace.

I would like to state my sincere gratitude to my supervisor, Lionel Briand, for giving me the opportunity to do a PhD within the SVV (Software Verification and Validation) laboratory and for his valuable comments and advice during our meetings. It was an honor to have him as a supervisor and to have had a chance to gain experience while working beside him. This work would not have gone far without his guidance, patience, enthusiasm, and encouragement.

I would like to express my infinite gratitude to, Arda Goknil, who has co-supervised this thesis. I have learned a lot from him both professionally and personally. His care and guidance were very constructive and rich so that they helped me to hone my writing and presentation skills.

I would like to express my infinite gratitude to, Fabrizio Pastore. His availability, comments, and corrections are a relevant lead to the success of this work. I would like to thank him for his insightful comments, his guidance, and support.

I am grateful to the members of my defence committee: Shiva Nejati, Fabrizio Pastore, Alexander Egyed, and Stefania Gnesi for dedicating time and effort to review this dissertation. I would like to thank further Alexander Egyed and Stefania Gnesi for taking the time to travel to Luxembourg to attend my defence, despite their busy schedules.

I am grateful to the members of IEE Luxembourg, particularly Thierry Stephany, for sharing his valuable time and insights with us. I would like to thank IEE for offering me the opportunity to conduct a case study in realistic settings during my thesis.

Last but not least, I would like to thank all my family members who kept encouraging me and supporting me from day one of this academic journey. In particular, I would like to thank my lovely husband and colleague, Ghanem Soltana, who always believed in me even when I had doubts. Finally, Thanks to all my friends and colleagues who made me feel that four years were much shorter than what I initially expected.

Fonds National de la
Recherche Luxembourg

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR10045046).

Contents

Contents **xi**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Context 1

1.2 Problem Statement 2

1.3 Research Questions 4

1.4 Research Methodology 5

1.5 Approach 6

1.6 Contributions 8

1.7 Outline of the Thesis 10

2 Background and Definitions **14**

2.1 Introduction 14

2.2 Requirements Engineering 14

2.2.1 Software Requirements 15

2.2.2 Requirements Engineering Framework 15

2.2.3 Requirements Artifacts 17

2.2.3.1 Goals 17

2.2.3.2 Scenarios 17

2.2.3.3 Solution-Oriented Requirements 18

2.3 Software Product Line Engineering 19

2.3.1 Software Product Line Framework 19

2.3.1.1 Domain Engineering 20

2.3.1.2 Application Engineering 21

2.3.2 Product Line Variability 22

2.4 Conclusion 24

3 Product Line Use Case and Domain Models **25**

3.1 Introduction 25

3.2 Motivation and Context 27
3.3 Related Work 30
3.4 Overview of Our Modeling Method 32
3.5 Capturing Variability in Requirements 33
3.5.1 Use Case Diagram with Product Line Extensions 33
3.5.2 Restricted Use Case Modeling (RUCM) and its Extensions 36
3.5.3 Domain Model and OCL Constraints 39
3.6 Tool Support 40
3.7 Industrial Case Study 42
3.8 Conclusion 44

4 Configuration of Product Specific Use Case and Domain Models 46
4.1 Introduction 46
4.2 Motivation and Context 48
4.3 Related Work 49
4.4 Overview of Our Approach 53
4.5 Configuration of Product Specific Use Case and Domain Models 55
4.6 Consistency Checking of Configuration Decisions 62
4.6.1 Objective and Assumptions 62
4.6.2 Propositional Logic Mappings 63
4.6.3 Consistency Checking Algorithm 65
4.6.4 Non-configurable PL Use Case Diagrams 69
4.7 Generation of Product Specific Use Case Models 70
4.8 Tool Support 74
4.8.1 Tool Architecture 74
4.8.2 Tool Features 76
4.9 Evaluation 79
4.9.1 Industrial Case Study 79
4.9.1.1 Goal 79
4.9.1.2 Study Context 80
4.9.1.3 Results 81
4.9.2 Questionnaire Study 82
4.9.2.1 Planning and Design 82
4.9.2.2 Results and Analysis 83
4.9.2.3 Threats to Validity 86
4.9.3 Discussions with the Analysts and Engineers 86
4.9.3.1 Modeling Effort 87
4.9.3.2 Degree of Automation 87
4.9.3.3 Limitations of the Configuration Approach 88
4.10 Conclusion 88

5 Change Impact Analysis for Evolving Configuration Decisions 91
5.1 Introduction . . . 91
5.2 Motivation and Context . . . 93
5.3 Related Work . . . 95
5.4 Overview of the Approach . . . 99
5.5 Identification of Change Impact on Decisions for PL Use Case Diagrams . . . 100
5.5.1 Identification of Subsequent Decision Restrictions . . . 105
5.5.2 Identification of Contradicting Decision Restrictions . . . 109
5.6 Incremental Reconfiguration of PS Use Case Models . . . 110
5.6.1 Model Matching and Change Calculation . . . 111
5.6.2 Regeneration of PS Use Case Models . . . 114
5.7 Tool Support . . . 116
5.7.1 Tool Architecture . . . 116
5.7.2 Tool Features . . . 118
5.8 Evaluation . . . 120
5.8.1 Questionnaire Study . . . 120
5.8.1.1 Planning and Design . . . 120
5.8.1.2 Results and Analysis . . . 121
5.8.2 Discussions with the Analysts and Engineers . . . 124
5.8.2.1 Industrial Adoption of the Approach . . . 125
5.8.2.2 Limitations of the Approach . . . 125
5.8.3 Industrial Case Study . . . 125
5.8.3.1 Goal . . . 126
5.8.3.2 Study Context . . . 126
5.8.3.3 Results and Analysis . . . 126
5.8.4 Threats to Validity . . . 127
5.9 Conclusion . . . 128

6 Automated Test Case Classification and Prioritization in Product Lines 130
6.1 Introduction . . . 130
6.2 Motivation and Context . . . 132
6.3 Related Work . . . 136
6.4 Overview of the Approach . . . 140
6.5 Classification of System Test Cases in a Product Family . . . 141
6.5.1 Steps 1 and 2: Model Matching and Change Calculation . . . 144
6.5.2 Step 3: Test Case Classification . . . 145
6.5.2.1 Use Case Model Generation . . . 146
6.5.2.2 Identification of Tested Use Case Scenarios . . . 148
6.5.2.3 Identification of Test Cases for Use Case Scenarios . . . 148
6.5.2.4 Impact Identification . . . 150
6.5.3 Step 4: Impact Report Generation . . . 156
6.6 Prioritization of System Test Cases in a Product Family . . . 157

6.7 Tool Support 160
6.7.1 Tool Architecture 160
6.7.2 Tool Features 162
6.8 Evaluation 162
6.8.1 Subject of the Study 163
6.8.2 Experiment Setup 164
6.8.3 Results 164
6.8.3.1 RQ1 164
6.8.3.2 RQ2 165
6.8.3.3 RQ3 166
6.8.3.4 RQ4 169
6.8.4 Threats to Validity 171
6.9 Conclusion 171

7 Conclusion 173
7.1 Introduction 173
7.2 Problems 173
7.3 Solutions 174
7.4 Future Research Directions 177

List of Papers 179

Bibliography 181

A Configuration Algorithm 202

B Algorithm for Identification of Subsequent Decision Restrictions 214

List of Figures

1.1 Clone-and-Own Reuse at IEE for the STO Product Family 2
1.2 Research Methodology 5
1.3 Overview of Our Solution 7
1.4 Thesis Map 11

2.1 The Requirements Engineering Framework [Pohl, 2010] 16
2.2 Software Product Line Engineering Framework [Pohl et al., 2005] 20
2.3 Variability Metamodel [Pohl et al., 2005] 23

3.1 Part of the UML Use Case Diagram for STO 27
3.2 Overview of PUM 32
3.3 Graphical Notation of Product Line Extensions for Use Case Diagrams 34
3.4 Extended UML Use Case Metamodel 35
3.5 Part of the Product Line Use Case Diagram for STO 36
3.6 Simplified Portion of the Domain Model for STO 39
3.7 Layered Architecture of PUM-C 41

4.1 Overview of the Approach 54
4.2 Decision Metamodel 56
4.3 Example Product Line Use Case Diagram 57
4.4 Generated Product Specific Use Case Diagram 58
4.5 Generated Product Specific Domain Model 61
4.6 Example Decision Model Resulting from the Example Configuration 62
4.7 Mapping from PL Use Case Diagram to Propositional Logic 64
4.8 An Example of a Non-Configurable PL Use Case Diagram 69
4.9 Overview of the PS Use Case and Domain Model Generation 71
4.10 Example Source and Target Patterns with Decisions for Use Case Diagrams 72
4.11 Example Source and Target Patterns with Decisions for Use Case Specifications 73
4.12 Layered Architecture of PUMConf 74
4.13 Menu to Activate IBM DOORS Plug-ins for PUMConf 75
4.14 PUMConf User Interface for Reporting Inconsistencies 77

4.15 PUMConf's User Interface for (a) listing variation points driving decisions, (b) selecting variant use cases for the selected variation point, and (c) showing the updated list of variation points after the decision. 78

4.16 PUMConf's User Interface for Selecting Optional Steps in Use Cases. 79

4.17 PUMConf's User Interface for (a) selecting variant use cases for the corresponding variation point, (b) explaining the contradicting decisions, (c) updating the contradicting decision, and (d) showing the updated list of variation points after the updated decision. . . 80

4.18 Responses to the Questions Related to the Product Line Use Case Modeling Method . . . 84

4.19 Responses to the Questions Related to the Configuration Approach 85

5.1 Part of an STO Use Case Specification with Trace Links 95

5.2 Overview of the Approach 100

5.3 An Example Product Line Use Case Diagram 101

5.4 Overview of the Model Differencing and Regeneration Pipeline 110

5.5 (a) Decision Metamodel, (b) Part of the Example Decision Model before Changes (*M1*), and (c) Part of the Example Decision Model after Changes (*M2*) 112

5.6 Regenerated Product Specific Use Case Diagram 115

5.7 Layered Architecture of PUMConf 117

5.8 PUMConf's User Interface for Proposing a Diagram Decision Change 118

5.9 PUMConf's User Interface for Displaying the Change Impact of Diagram Decision Changes on Other Diagram Decisions 119

5.10 Responses to the Questions Related to the Configuration Approach 122

5.11 Responses to the Questions Related to the Change Impact Analysis Approach 123

6.1 Opportunistic Reuse of System Test Cases 133

6.2 Test Cases derived from (a) the Basic Flow of the Use Case *Recognize Gesture* in Table 6.1 and (b) the First Specific Alternative Flow of the Same Use Case 134

6.3 Overview of the Approach 141

6.4 Overview of the Model Differencing and Test Case Classification Pipeline 142

6.5 (a) Decision Metamodel, (b) Part of the Example Decision Model of the Previous Product (M_i), and (c) Part of the Example Decision Model of the New Product (M_{new}) 143

6.6 Metamodel for Use Case Scenario Models 146

6.7 Use Case Scenario Models Generated from the Use Case Specifications in Table 6.1 . . . 147

6.8 Some Tested Use Case Scenarios 149

6.9 (a)Traceability Metamodel and (b) Example Model 150

6.10 Two New Scenarios Derived from the Scenarios in Fig. 6.8 153

6.11 PUMConf's User Interface for Guidance 155

6.12 System Test Case derived from the System Test Case in Fig. 6.2(a) of the Scenario in Fig. 6.8(a) 156

6.13 Overview of the Test Case Prioritization Pipeline 158

6.14 Layered Architecture of PUMConf 161

6.15 Prioritization results: percentage of failures detected after running *x* prioritized test cases. 170

List of Tables

1.1 Mapping the Research Questions to the Chapters of the Thesis 13

3.1 Some Use Cases for STO 29
3.2 Some STO Use Cases in the extended RUCM 37
3.3 Some OCL Constraints for STO Use Cases 40
3.4 Product Line Use Cases in the Case Study 42
3.5 Size of the Domain Model 42

4.1 Some Example Specifications of the PL Use Cases in Fig. 4.3 59
4.2 Some of the Generated PS Use Case Specifications 60
4.3 Results Summary for the Configuration of STO Use Case and Domain Models for Various Car Manufacturers 82

5.1 Change Types for Diagram Decisions 101
5.2 Example Decisions for the PL Use Case Diagram in Fig. 5.3 102
5.3 Restrictions Inferred from the Example Decisions in Table 5.2 104
5.4 Reconfigured Product Specific Specifications 116
5.5 Decision Change Scenarios 127
5.6 Summary of the Reconfiguration of the PS Use Case Models for STO 128

6.1 Some of the Generated Product Specific Use Case Specifications for STO 135
6.2 Matching Decisions in M_i and M_{new} in Fig. 6.5 144
6.3 Changes in Use Case Scenarios and Classification of System Test Cases 152
6.4 Excerpt of the Training Data Set used for Logistic Regression 159
6.5 Overview of the STO Product Line Use Cases 163
6.6 Details of the Configured Products in the STO Product Line 164
6.7 Test Case Classification Results for Single-Product Settings 165
6.8 Test Case Classification Results for Whole-Line Settings 166
6.9 Relevance of Scenarios Identified using Single-Product Settings 167
6.10 Relevance of Scenarios Identified using Whole-Line Settings 167
6.11 Analysis of Significant Factors identified by Logisitic Regression 168
6.12 Test Case Prioritization Results 168
6.13 Test Development Costs Savings 169
6.14 Development Process Savings 169

List of Algorithms

1 Part of config . . . 66

2 checkRequiredUC . . . 68

3 impact . . . 103

4 inferDecisionRestrictions . . . 106

5 inferConflictingUC . . . 107

6 checkDecisionRestrictions . . . 109

7 Algorithm for Steps 1 and 2 in Fig. 5.4 . . . 113

8 Test Case Classification Algorithm . . . 145

9 Algorithm for analyzeImpact . . . 151

10 Algorithm for identifyNewScenarios . . . 154

11 config . . . 204

12 config(continued) . . . 205

13 checkConflictingVP . . . 206

14 checkConflictingUC . . . 207

15 checkRequiringVP . . . 208

16 checkRequiredVP . . . 209

17 checkRequiringUC . . . 210

18 checkRequiredUC . . . 211

19 inferSelectedElements . . . 212

20 inferUnselectedElements 213

21 inferDecisionRestrictions 215

22 inferRequiredByUC 216

23 inferConflictingUC 217

24 inferRequiredByVP 218

25 inferConflictingVP 219

26 inferRequiringVP 220

27 inferRequiringUC 221

28 checkDecisionRestrictions 222

Chapter 1

Introduction

In this chapter, we describe the problem addressed in this thesis, together with our contributions and an outline of the thesis.

1.1 Context

In various domains such as automotive and avionics, software systems are quickly getting larger and more complex. These systems often consist of various interacting subsystems, e.g., lighting systems, engine controller, and sensing systems. Further, many suppliers are typically involved in system development. Each supplier is mostly specialized in developing one or two of these subsystems. For example, in the automotive domain, while one supplier provides a sensing system monitoring the driver seat for occupancy by means of a pressure sensitive sensor, another supplier may develop an engine controller using the output of the sensing system to prevent unintentional vehicle starts and unnecessary fuel consumption. These suppliers develop multiple versions of the same product since they work with many manufacturers (customers). Therefore, given the complexity arising from the context described above, systematic and supported Product Line Engineering (PLE) is crucial in their software development practice, from requirements analysis to implementation and testing.

This thesis was motivated by discussions with IEE S.A. (in the following "IEE") [IEE, 2018], a leading supplier in automotive sensing systems enhancing safety and comfort in vehicles produced by major car manufacturers worldwide. Such systems monitor the physical environment by means of physical components (e.g., electrical field sensors, pressure sensitive sensors, and force sensing resistors), detect events or changes regarding objects and humans (e.g., seat occupant classification, gesture recognition, and driver presence detection), and provide the corresponding output to other subsystems (e.g., airbag control unit, trunk controller, and engine controller). At IEE, similar to many other development environments, use cases (including use case diagrams and use case specifications) are the main artifacts employed to elicit requirements and communicate with customers. In order to clarify the terminology used in the requirements and to provide a common understanding of the application domain concepts, use cases are often accompanied by a domain model formalizing concepts and their relationships, often under the form of a class diagram with constraints. The system test cases are derived from the use case models to validate the system developed for the customer.

We use *Smart Trunk Opener (STO)* as a case study, to motivate and assess the techniques and tools developed in this thesis. STO is a real-time automotive embedded system developed by IEE. It provides automatic, hands-free access to a vehicle's trunk, in combination with a keyless entry system. In possession of the vehicle's electronic remote control, the user moves her leg in a forward and backward direction at the vehicle's rear bumper. STO recognizes the movement and transmits a signal to the keyless entry system, which confirms that the user has the remote. This allows the trunk controller to open the trunk automatically.

In the remainder of the present chapter, we introduce change management in product lines for use case-driven development and testing. In the next section, the problems this thesis addresses are explained. Research objective and research questions related to the problem statement are given in Section 1.3. Section 1.4 presents the research methodology that we follow in this thesis. Our solution approach is described and the contributions of this thesis are introduced in Section 1.5 and Section 1.6. Finally, we provide the outline of the thesis in Section 1.7.

1.2 Problem Statement

The work presented in this dissertation took place in the context of embedded software systems, interacting with multiple other external systems, and developed according a use case-driven process, by a supplier for multiple manufacturers (customers). In such a context, requirements variability is communicated to customers and an interactive configuration process is followed for which guidance and automated support are needed. For instance, for each product in a product family, IEE negotiates with customers how to resolve variation points in requirements, in other words how to configure the product line.

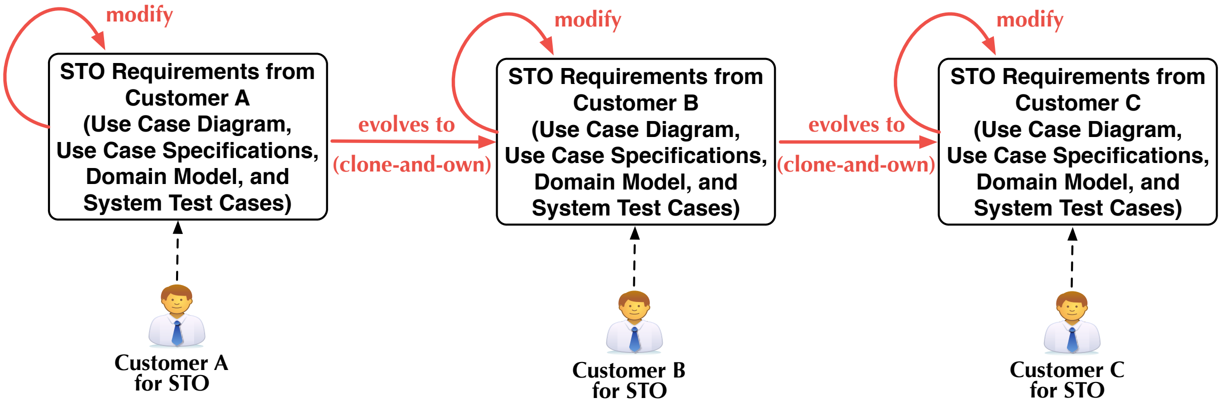

modify
STO Requirements from Customer A (Use Case Diagram, Use Case Specifications, Domain Model, and System Test Cases)
evolves to
(clone-and-own)
modify
STO Requirements from Customer B (Use Case Diagram, Use Case Specifications, Domain Model, and System Test Cases)
evolves to
(clone-and-own)
modify
STO Requirements from Customer C (Use Case Diagram, Use Case Specifications, Domain Model, and System Test Cases)
Customer A for STO
Customer B for STO
Customer C for STO

Figure 1.1. Clone-and-Own Reuse at IEE for the STO Product Family

The current use case-driven development and testing practice at IEE, like in many other environments, is based on clone-and-own reuse [Clements and Northrop, 2001] (see Fig. 1.1). IEE starts a project with an initial customer. The product requirements are elicited from the initial customer and documented as a use case diagram, use case specifications, and a domain model. For each new customer in the product family, IEE analysts need to clone the current models, and negotiate variabilities with the customer to produce a new use case diagram, set of specifications, and domain model (see *clone-and-own* in Fig. 1.1). As a result of the negotiations, IEE analysts make changes in the cloned models (see *modify*). They derive and select new system test cases from the modified use cases (i.e., regression test selection for system test cases). With such practice, variants and variation points (i.e., where potential changes are made) are not documented and IEE analysts, together with the customer, need to evaluate the entire use cases, domain model and system test cases.

The benefits of automated use case-driven configuration have been acknowledged and there are proposed approaches in the literature [Alves et al., 2010, Rabiser et al., 2010, Alférez et al., 2014]. Many studies [Eriksson et al., 2009, Czarnecki and Antkiewicz, 2005, Alférez et al., 2009] provide configuration approaches which require that feature models be traced as an orthogonal model to artifacts such as UML use case, activity and class diagrams, and system test cases. In order to employ these approaches in industrial practice, the analysts need to provide feature models with their traces to use cases and related artifacts. The evolution of feature models also requires these traces to be maintained manually by the analysts. In addition, none of these approaches support the evolution of requirements in the context of product configuration. For instance, Eriksson et al. [Eriksson et al., 2009] provide an approach to manage natural-language requirements specifications in the software product line context. Variability is captured and managed using a feature model while requirements in various forms, e.g., use cases, textual requirements specifications and domain model, are traced to the feature model. Moon et al. [Moon et al., 2005, Moon and Yeom, 2004] propose a method that generates Product Specific (PS) use cases from Product Line (PL) use cases without using any feature model. However, the proposed method requires Primitive Requirements (PR) (i.e., building blocks of complex requirements) to be specified by the analysts and traced to the use case diagram and specifications via *PR - Context* and *PR - Use Case* matrices. The configuration takes place by selecting the *PR*s in the matrices without any automated guidance. Wang et al. [Wang et al., 2017] [Wang et al., 2016] [Wang et al., 2013] propose a methodology for the selection of test cases for products belonging to product families. The methodology adapts the formalisms commonly used to model product lines, i.e., feature models and component family models, to support test case selection. Feature models are used to capture all the choices that affect test case selection while test cases are manually traced to these feature models. Due to deadline pressure and limited resources, many software development companies find such additional traceability and maintainability effort to be impractical.

Below we present an overview of the problems addressed by this thesis:

- **Modeling Variability in Requirements with Additional Traceability to Feature Models.** Analysts need to explicitly document variability information (e.g., variant use cases, variation points, and optional steps) for use cases and a domain model to be communicated to customers during product configuration. Relating feature models to use cases and domain model is the most straightforward option but has shortcomings in terms of additional modeling and traceability effort. For example, it would not be easy for analysts and customers to comprehend and visualize all variability information traced to use case diagram, use case specifications and domain model. In STO, we identified 15 mandatory and 14 variant use cases which contain 8 variation points and 7 variant dependencies. The STO use cases include 244 use case flows (29 basic flows, 202 alternative flows, and 13 optional alternative flows) with 27 optional steps while the STO domain model contains 11 variant domain entities. The variability information scattered across all these use case flows with trace links from feature models would be communicated to customers and used to configure a product.

- **Manual, Expensive and Error Prone Configuration of Product Specific Requirements.** In order to facilitate use case-driven configuration in industrial practice, a high degree of automation is required to interactively guide analysts in their decisions. Before configuration, it should be automatically confirmed that all artifacts with variability information, including use case diagram, specifications, and domain model, are consistent. Any inconsistency in these artifacts may cause invalid configuration outputs. Adding to the complexity affecting the decision-making process during configuration, there may be contradicting decisions and dependencies among decisions. Changes on configuration decisions may impact prior decisions as well as subsequent decisions. During the configuration process, the analysts need to be interactively informed about contradicting decisions, the order of possible decisions, and the impact of decision changes on other decisions. Without interactive guidance and proper tool support, the analysts have to manually identify and fix inconsistent PL artifacts, resolve decision contradictions, and change further decisions, which leads to time-consuming, expensive and error prone configuration of PS use case and domain models.

- **Manual and Expensive Regression Testing in Product Families.** The current use case-driven testing practice in many software development environments follows the testing strategy referred to as *opportunistic reuse of test assets* [de Mota Silveira Neto et al., 2011] for product families. When there is an initial customer for a product in the product family, the product requirements are elicited from the initial customer and documented as a use case diagram and use case specifications. System test cases are then generated from the use cases for the initial customer. For each new customer in the product family, test engineers manually choose and prioritize, from the existing test suite(s) for the initial/previous product(s), test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly in the new product. This form of test reuse is not performed systematically, which means that there is no structured, automated method that supports the activity of selecting and prioritizing test cases. The current practice is fully manual, error-prone and time-consuming, which leads to ad-hoc change management for system test cases in product lines. Therefore, product line modeling and testing techniques are needed to automate the reuse of system test cases in the context of use case-driven development of a product family.

1.3 Research Questions

The objective of this thesis is to investigate to what extent and how product line engineering techniques can be used to support change management for use case models and system test cases in product families by enhancing automation in use case-driven configuration and regression testing. Within the context of this objective, we provide a configuration framework for use case models and system test cases in product families. A number of research questions need to be answered. Answering these questions will give us a better insight about the problem domain and the deficiencies of the current solutions.

- *Research Question 1:* What are the key concepts for managing requirements? What are the core concepts and techniques for managing variability in product families?
- *Research Question 2:* How to model variability in use case and domain models without additional traceability to feature models? Are there any existing techniques for use case and domain models to model variability? Which techniques can be used?
- *Research Question 3:* To what extent and how can we automate the interactive configuration of use case and domain models? How can we support the analysts for making configuration decisions and for generating PS use case and domain models?
- *Research Question 4:* What are the change scenarios for use case models and system test cases in a product family? What is necessary for these change scenarios to be handled in the configuration process? Which solutions can be used?
- *Research Question 5:* How can a change in a configuration decision be propagated to other decisions in PL use case models and to system test cases? How can we support the analysts in performing changes? How can we reconfigure PS use case and domain models for decision changes? How can we select and prioritize system test cases for such changes?

These questions guide our research in this thesis. In Section 1.7, we give the outline of the thesis and a table that relates the research questions to the chapters in which we provide answers to the questions (see Table 1.1).

1.4 Research Methodology

In this thesis, we try to solve two kinds of problems: *design problems*, i.e., the difference between the current and desired states of the world, and *knowledge problems*, i.e., the difference between the current and desired knowledge states [Wieringa et al., 2006] [Wieringa, 2009] [Wieringa, 2010].

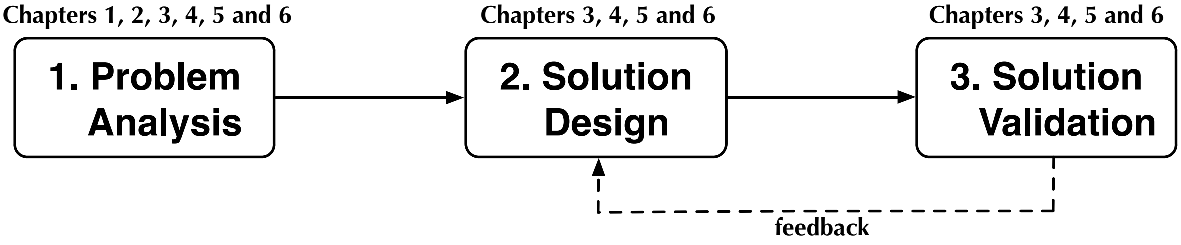


Figure 1.2. Research Methodology

Our research methodology has three steps: *problem analysis*, *solution design* and *solution validation* (see Fig. 1.2).

In the first step, we identify our design problem, i.e., the difference between the current and desired states of software development practice in the context of change management for use case-driven development and testing in product families. Then, we solve a knowledge problem, for instance, we want to understand what the published change management approaches for product line use case models are and what the current deficiencies of those current approaches are. For that purpose, we analyze the literature from different research areas (i.e., requirements engineering and product line engineering) to discover possible change management problems in current product line approaches for use case-driven development and testing.

In the second step, the results of the first phase are employed to design a new solution. We provide a change management framework for product lines within the context of use case-driven development and testing. Our goal is to improve change management for use case models and system test cases in a product family by providing a use-case driven configuration approach supporting change impact analysis and regression testing.

Finally, in the third step, we validate our solution by investigating its availability in real industrial settings for the problems discovered in the problems analysis phase. This is a knowledge problem since we want to gain knowledge about the properties of our solution, and the relation between the solution and the problems. The output of the third step is fed back to the second step to improve the solution.

1.5 Approach

We choose Product Line Engineering (PLE) as a solution platform for our approach. PLE provides a way to engineer a set of related products as a product family in an efficient manner, taking advantage of the products' similarities (commonalities) while managing their differences (variabilities). It enables engineers to configure products in a product family by making configuration decisions about the product variabilities. We employ Natural Language Processing (NLP) techniques to process and analyze use cases. To manage changes in use case models, domain models and system test cases in product families, we provide a use case-driven configuration framework within the context of PLE. Fig. 1.3 presents an overview of our proposed framework.

To capture variability and commonality information in use case and domain models, we propose to use some existing product line extensions of use case and domain models with some further extensions we design. Using these extensions, the analyst produces three artifacts: a Product Line (PL) use case diagram, PL use case specifications, and a PL domain model (see *Elicit Product Line Use Case Models* in Fig. 1.3). It may not always be possible to model PL use cases without starting from product use cases; the analyst elicits use cases of a specific product, and then identifies variabilities and commonalities for the product family. We observe that most of the projects in industry start with an initial customer for which the product is designed and produced. Other potential customers are typically engaged after the release of the initial product. At this phase of product development, the analyst starts identifying commonalities and variabilities of the product family based on the use cases of the initial product.

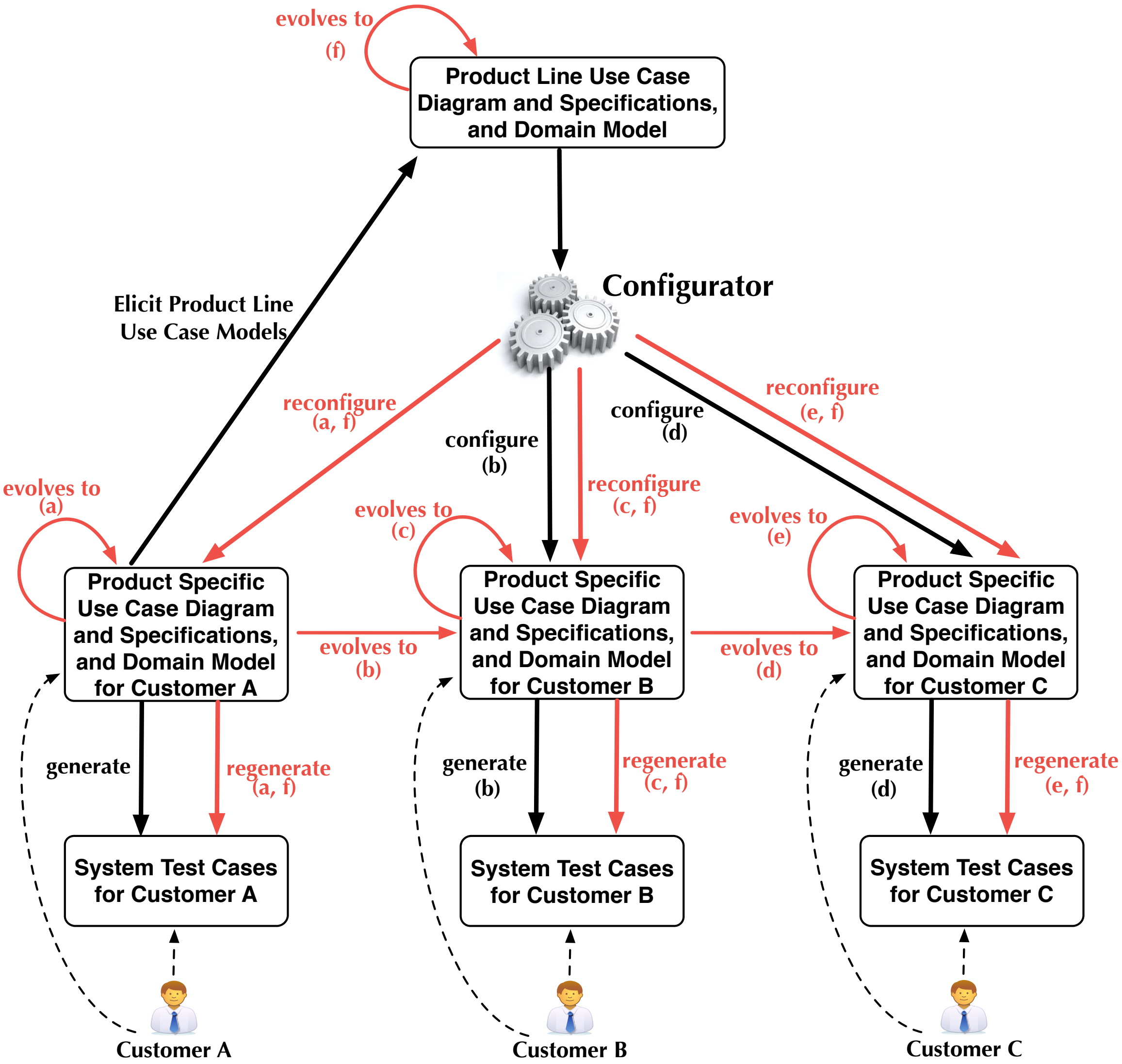

evolves to
(f)
Product Line Use Case Diagram and Specifications, and Domain Model
Configurator
Elicit Product Line Use Case Models
reconfigure (a, f)
configure (b)
reconfigure (c, f)
configure (d)
reconfigure (e, f)
evolves to (a)
evolves to (c)
evolves to (e)
Product Specific Use Case Diagram and Specifications, and Domain Model for Customer A
Product Specific Use Case Diagram and Specifications, and Domain Model for Customer B
Product Specific Use Case Diagram and Specifications, and Domain Model for Customer C
evolves to (b)
evolves to (d)
generate
regenerate (a, f)
generate (b)
regenerate (c, f)
generate (d)
regenerate (e, f)
System Test Cases for Customer A
System Test Cases for Customer B
System Test Cases for Customer C
Customer A
Customer B
Customer C

Figure 1.3. Overview of Our Solution

In order to facilitate use case-driven configuration in industrial practice, we develop a use case-driven configurator providing a high degree of automation while the analysts are interactively guided in their configuration decisions. With interactive guidance and proper tool support, the analysts make decisions and resolve contradictory decisions, which leads to the automatic generation of Product Specific (PS) use case and domain models. With new customers (see ‘evolves to’ in *b* and *d* in Fig. 1.3), the analyst is asked to input configuration decisions regarding variation points captured in PL use case and domain models to automatically configure the product line into a product (see ‘configure’ in *b*, and *d* in Fig. 1.3). The configuration of PS use case and domain models is an automated, iterative, and interactive decision-making activity. When a decision is made, the consistency of the decision with prior decisions is checked by our configurator. There might be contradicting decisions in the PL use case diagram such as two decisions resulting in selecting variant use cases violating some dependency constraints. These are automatically determined and reported *a posteriori* and the analysts can backtrack and revise their decisions.

Configuration decisions may frequently change, resulting in the reconfiguration of PS use case models (see 'evolves to' in *a*, *c* and *e* in Fig. 1.3). Impacted decisions, i.e., subsequent decisions to be made and prior decisions cancelled or contradicting when a decision changes, need to be identified to incrementally reconfigure the generated use case models (see 'reconfigure' in *a*, *c* and *e* in Fig. 1.3). To this end, we develop a change impact analysis approach to identify impacted configuration decisions and parts of PS models that need to be reconfigured when the PL models and the configuration decisions evolve.

When configuration decisions evolve in a product family (see 'evolves to' in *a*, *c*, and *e* in Fig. 1.3), the change impact on the execution of system test cases derived from these requirements (see 'regenerate' in *a*, *c*, and *e* in Fig. 1.3) need to be assessed. We develop a regression testing approach that automatically chooses and prioritizes, from an existing test set, test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly.

Reconfiguration is needed not only for changes in configuration decisions (see 'evolves to' in *a*, *c*, and *e* in Fig. 1.3) but also for changes in PL use case models (see 'evolves to' in *f*). The former requires reconfiguration only for the product concerned with the decisions, while the latter needs an impact assessment method to analyze change impact on PL use cases before the reconfiguration of use case models for all products in the product family (see 'reconfigure' in *f* in Fig. 1.3). When the PL use case and domain models evolve, the change impact on the execution of system test cases derived from the configured use case and domain models (see 'regenerate' in *f* in Fig. 1.3) also need to be assessed. Currently, our framework does not support it. As future work, we plan to provide a change impact analysis and automated regression testing approach for changes in PL use case and domain models (see Chapter 7).

1.6 Contributions

This thesis provides the following contributions:

- A modeling method for capturing variability information in Product Line (PL) use case and domain models.

Chapter 3 presents the Product line Use case modeling Method (PUM), which enables the analysts to capture and document variability in PL use case diagrams, use case specifications, and domain model. For PL use case diagrams, we employ the diagram extensions proposed by Halmans and Pohl [Halmans and Pohl, 2003]. These extensions overcome the shortcomings of textual representations of variability, such as implicit variants and variation points. Further, for PL use case specifications, we employ Restricted Use Case Modeling (RUCM) [Yue et al., 2013], which includes a template and restriction rules to reduce imprecision and incompleteness in use cases. RUCM was a clear choice since it reduces ambiguity and facilitates automated analysis of use cases [Yue et al., 2011] [Yue et al., 2015b] [Wang et al., 2015a]. However, since it was not originally meant to model variability, we introduced some PL extensions to capture variability in use case specifications. To be able to capture variability in PL domain models, we rely on the stereotypes (i.e., *variation*, *variant* and *optional*), proposed by Ziadi and Jezequel [Ziadi and Jezequel, 2006] for UML class diagrams.

- An approach for automated configuration of Product Specific (PS) use case and domain models.

Chapter 4 presents a use case-driven configuration approach based on PUM. Our goal is to provide a degree of configuration automation that enables effective product-line management in use case-driven development, without requiring additional modeling artifacts and traceability effort. Our approach supports four activities. First, the analysts model the variability information explicitly in a PL use case diagram, its use case specifications, and its corresponding domain model. Second, the consistency of the PL use case diagram and specifications are checked and inconsistencies are reported if there are any. For instance, a variation point in the use case diagram might be missing in the corresponding use case specification or a use case specification may not conform to the extended RUCM template. Third, the analyst is guided to make configuration decisions based on variability information in the PL models. The partial order of decisions to be made is automatically identified from the dependencies among variation points and variant use cases. In the case of contradicting configuration decisions, such as two decisions resulting in selecting variant use cases violating some dependency constraints, we automatically detect and report them. The analyst must then backtrack and revise the decisions to resolve these inconsistencies. Fourth, based on configuration decisions, the PS use case and domain models are generated from the PL use case and domain models. To support these activities, we developed a tool, *PUMConf (Product line Use case Model Configurator).* The tool automatically checks the consistency of the PL models, identifies the partial order of decisions to be made, determines contradicting decisions, and generates PS use case and domain models.

- A change impact analysis approach for evolving configuration decisions in PL use case models.

Chapter 5 presents a change impact analysis approach, based on our use case-driven modeling and configuration techniques, to support the evolution of configuration decisions. We do not address here evolving PL use case models, which need to be treated in a separate approach. Change impact analysis provides a sound basis to decide whether a change is worth the effort and which decisions should be changed as a consequence [Passos et al., 2013]. In our context, we aim to automate the identification of decisions impacted by changes in configuration decisions on PL use case models. Our approach supports three activities. First, the analyst proposes a change but does not apply it to the corresponding configuration decision. Second, the impact of the proposed change on other configuration decisions for the PL use case diagram is automatically identified. In the PL use case diagram, variant use cases and variation points are connected to each other with some dependencies, i.e., *require*, *conflict* and *include*. In the case of a changed diagram decision contradicting prior and/or subsequent diagram decisions, such as a subsequent decision resulting in selecting variant use cases violating some dependency constraints because of the new/changed decision, we automatically detect and report them. To this end, we improved our consistency checking algorithm given in Chapter 4, which enables reasoning on subsequent decisions as part of our impact analysis approach. The analyst is informed about the change impact on decisions for the PL use case diagram. Based on this, the analyst should decide whether the proposed change is to be applied to the corresponding decision. Third, the PS use case models are incrementally regenerated only for the impacted decisions after the analyst actually makes all the required changes. To do so, we implemented a model differencing pipeline which identifies decision changes to be used in the reconfiguration of PS models. There are two sets of decisions: (i) the set of previously made decisions used to initially generate the PS use case models and (ii) the set of decisions including decisions changed after the initial generation of the PS models. Our approach compares the two sets to determine for which decisions we need to incrementally regenerate the PS models. To support these three activities, we extended our configurator, PUMConf.

- An approach for automated classification and prioritization of system test cases in a family of products.

Chapter 6 presents an automated classification and prioritization approach, based on our use case-driven modeling and configuration techniques, to support product line testing for evolving products of a product family in terms of evolving configuration decisions in PL use case model. In our context, we aim to automate the identification of system test cases impacted by changes in configuration decisions in PL use case models when a new product is configured in the product family. The initial product is tested individually and the following products are tested using regression testing techniques, i.e., test case selection and prioritization based on configuration decision changes between the previous product(s) and the new product to be tested. Our approach supports two activities. First, the system test cases of the previous product(s) are automatically classified as *obsolete*, *retestable*, and *reusable*. An *obsolete* test case cannot be executed on the new product as the corresponding use case scenarios are not selected for the new product. A *retestable* test case is still valid but needs to be rerun to determine the possible impact of changes whereas a *reusable* test case is also valid but does not need to be rerun for the new product. We also identify the use cases of the new product that have not been tested so far in the product family. To do so, we reused our model differencing pipeline in Chapter 5 which identifies decision changes to be used in the reconfiguration of PS models. Our approach classifies the configuration decisions as *new*, *deleted* and *updated* to identify the impacted parts of the use case models of the previous product(s). By using the traces from the impacted parts of the use case models to system test cases, we automatically classify the system test cases for test case selection. Second, the system test cases are automatically prioritized based on multiple risk factors such as fault proneness of requirements and requirements volatility in the product line. To this end, we developed a prediction model that computes a prioritization score for each system test case based on these factors. To support these activities, we extended our configurator, PUMConf.

1.7 Outline of the Thesis

Fig. 1.4 shows the map of the thesis with chapters and relations among them.

The thesis consists of the following chapters:

- **Chapter 2 Backround and Definitions.** This chapter describes the concepts used in the thesis. It introduces concepts and techniques from the areas of Requirements Engineering and Product Line Engineering as they are described in the literature.

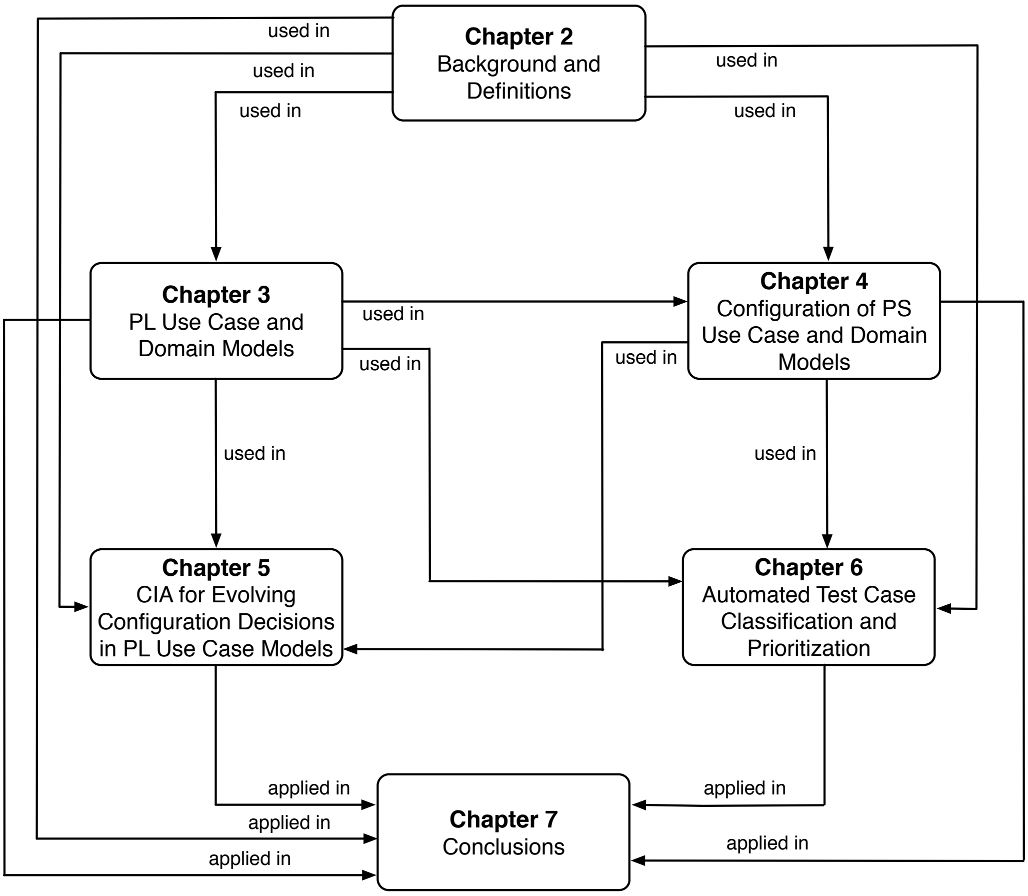

Chapter 2
Background and Definitions
Chapter 3
PL Use Case and Domain Models
Chapter 4
Configuration of PS Use Case and Domain Models
Chapter 5
CIA for Evolving Configuration Decisions in PL Use Case Models
Chapter 6
Automated Test Case Classification and Prioritization
Chapter 7
Conclusions
used in
applied in

Figure 1.4. Thesis Map

- **Chapter 3 Product Line Use Case and Domain Models.** This chapter proposes, applies, and assesses the Product line Use case modeling Method (PUM) to support variability modeling in PL use case diagrams, specifications, and domain models, without making use of feature models, thus avoiding unnecessary modeling overhead. We present the tool support for PUM relying on Natural Language Processing (NLP) to check the consistency of PL use case and domain models. We illustrate PUM in an industrial automotive embedded system, i.e., STO, and report lessons learned and results from structured interviews with experienced engineers. This chapter is an enhancement of the results published in [Hajri et al., 2015].

- **Chapter 4 Configuration of Product Specific Use Case and Domain Models.** This chapter proposes, applies, and assesses our use case-driven configuration approach based on PUM. For given PL use case and domain models, the configuration approach checks the consistency of the models, interactively receives configuration decisions from analysts, automatically checks decision consistency, and generates PS use case and domain models from the PL models and decisions. The algorithm for consistency checking of configuration decisions is presented in this chapter. We also present the features of PUMConf for consistency checking of decisions and for generation of PS use case and domain models. We evaluate our approach in an industrial case study, i.e., STO, which shows evidence that it is practical and beneficial to capture variability at the appropriate level of granularity and to configure PS use case and domain models in industrial settings. This chapter is an enhancement of the results published in [Hajri et al., 2018b] and [Hajri et al., 2016].

- **Chapter 5 Change Impact Analysis for Evolving Configuration Decisions.** This chapter proposes, applies, and assesses a change impact analysis approach for evolving configuration decisions in PL use case models. We present the algorithm for identifying the impact of decision changes on prior and subsequent decisions. The model differencing and regeneration pipeline for incremental reconfiguration of PS use case models is also presented in this chapter. We evaluate our approach in an industrial case study, i.e., STO, which provides evidence that it is practical and beneficial to analyze the impact of decision changes and to incrementally regenerate PS models in industrial settings. This chapter is an enhancement of the results published in [Hajri et al., 2017] and [Hajri et al., 2018a].

- **Chapter 6 Automated Test Case Classification and Prioritization in Product Lines.** This chapter proposes, applies and assesses a test selection and prioritization approach to support product line testing for evolving products of a product family in terms of evolving configuration decisions in PL use case models. We evaluate our approach in an industrial case study, which provides evidence that it is practical and beneficial to select and prioritize system test cases for a new product in industrial settings. Our algorithm for classifying system test cases for product-to-product testing in a product family is presented in this chapter. We also present our prediction model that computes a prioritization score for each system test case based on multiple risk factors.

- **Chapter 7 Conclusions.** This chapter provides conclusions and an evaluation of the contributions in this thesis, and describes directions for future work.

Table 1.1 relates the research questions to the chapters in which we provide answers to these questions.

Table 1.1. Mapping the Research Questions to the Chapters of the Thesis

	Chapter						
	1	2	3	4	5	6	7
Research Question 1		+					
Research Question 2			+				
Research Question 3				+	+		
Research Question 4					+	+	
Research Question 5					+	+	

Chapter 2

Background and Definitions

In our work, we utilize concepts and techniques from the areas of requirements engineering and product line engineering. In this chapter, we provide background information on these areas and introduce a set of definitions throughout the thesis.

2.1 Introduction

This chapter gives an overview of the concepts and terms used in this thesis. Since various definitions of these concepts are found in the literature, we try to select a consistent set of definitions that support the understanding of the thesis.

In this chapter, we answer Research Question 1 (*What are the key concepts for managing requirements? What are the core concepts and techniques for managing variability in product families?*) as formulated in Chapter 1. Using the definitions in this chapter, we address software product line engineering within the context of requirements engineering and software testing.

This chapter is structured as follows. Section 2.2 describes the fundamentals of requirements engineering such as requirements elicitation and documentation. Section 2.3 focuses on software product line engineering and its application to use case driven development and testing.

2.2 Requirements Engineering

Requirements engineering refers to the process of specifying, analyzing, documenting and maintaining requirements in the engineering design process [Nuseibeh and Easterbrook, 2000]. Sommerville [Sommerville, 2009] describes requirements engineering as “the process of finding out, analyzing, documenting and checking the services and constraints for the system to be built”. According to Van Lamsweerde [van Lamsweerde, 2009], requirements engineering is “a coordinated set of activities for exploring, evaluating, documenting, consolidating, revising and adapting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies".

In this section, we start with the presentation of the key terminology and concepts in requirements engineering. We introduce a requirements engineering framework which represents the building blocks of requirements engineering. We then present the approaches for software requirements specification and documentation.

2.2.1 Software Requirements

There are various definitions and classifications of the term "requirement". It is defined in the IEEE 610.12-1990 standard [IEE, 1990] as: (a) a system capability needed by a system user to solve a problem or achieve an objective, (b) a capability that must be met by a system to satisfy a contract, standard or specification, and (c) a documented representation of a capability as in (a) or (b). "The description of the services and constraints for the system" is called a requirement by Sommerville [Sommerville, 2009], while the term "requirement" is defined as " a property which must be exhibited by a system" in SWEBook [SWE, 2014]. We use the definition in the IEEE 610.12-1990 standard [IEE, 1990] as our working definition for requirements in the thesis.

There is a terminology to distinguish different types of requirements such as user requirements, system requirements, software requirements, and functional / non-functional requirements. For instance, the term "user requirement" is used for high-level abstract requirements, while the term "system requirement" is used for the detailed descriptions of what the system should do [Sommerville, 2009]. We consider software requirements and software system requirements as synonyms and as a specialization of system requirements for software systems in this thesis. Software requirements are often classified as functional, non-functional and constraints (see [Pohl, 2010] [Sommerville, 2009] [Lauesen, 2002] [Robertson and Robertson, 2006] [Wiegers and Beatty, 2013]).

- **Functional Requirements.** Functional requirements describe services the system should provide. They may also state what the system should not do.
- **Non-Functional Requirements.** They are often called "quality attributes" of a system such as security, reliability, performance, maintainability, scalability, and usability.
- **Constraints.** They are organizational or technological requirements that restrict the way in which the system is developed [Robertson and Robertson, 2006] [Pohl, 2010].

2.2.2 Requirements Engineering Framework

The requirements engineering framework introduced by Pohl [Pohl, 2010] represents the key elements of a requirements engineering process. The framework consists of the following building blocks (see Fig. 2.2):

- **System context:** Software system requirements are heavily influenced by the context of the system under development. The system context consists of various aspects that are related to

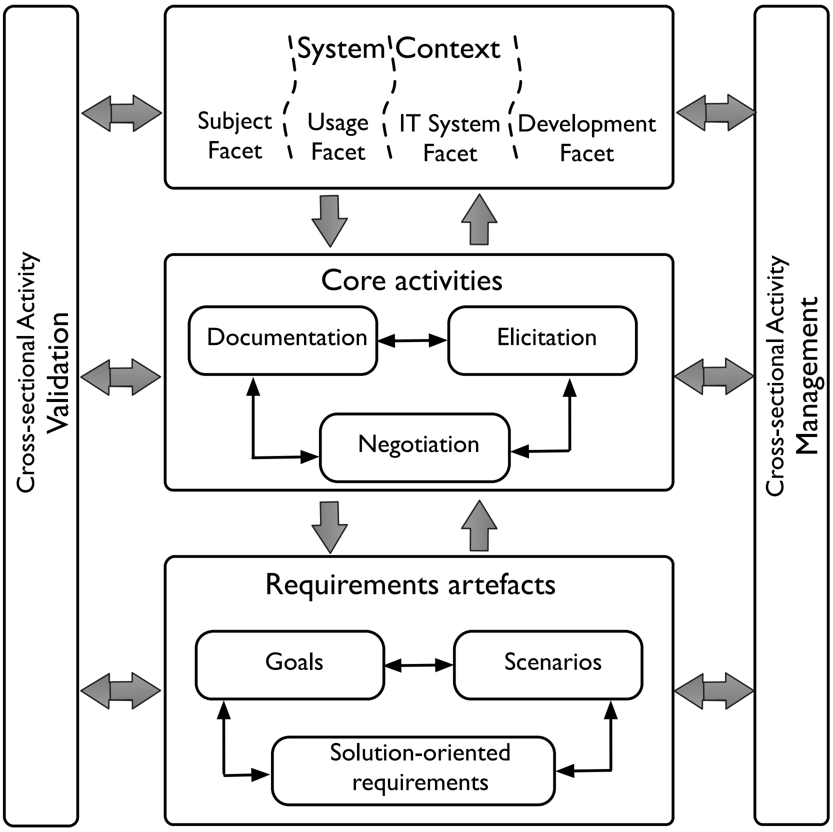


Figure 2.1. The Requirements Engineering Framework [Pohl, 2010]

the system to be developed, such as business processes, hardware and software components, third-party systems, standards and stakeholders [Pohl, 2010]. It is structured into four facets: *subject facet*, i.e., all objects and events in the system context, *usage facet*, i.e., all aspects that are related to the system usage, *IT system facet*, i.e., all aspects of the operational and technical environment including policies, strategies and guidelines, and *development facet*, i.e., all aspects of the context that are related to the development processes of the system.

- **Core requirements engineering activities:** The core activities concern the understanding of the requirements, the documentation and specification of the elicited requirements, and the identification and resolution of conflicts between various stakeholders.
- **Cross-sectional activities:** These activities support the core activities and secure the results of requirements engineering. The validation activity aims at detecting defects in the requirements, checking the compliance between the core activities, and validating whether the core activities have been properly followed. The management activity comprises the management of the requirements artifacts, the planning and control of the core requirements engineering activities, and the identification of changes in the system context.
- **Requirement artifacts:** The requirements engineering framework distinguishes three main artifacts: *goal*, *scenarios*, and *solution-oriented requirements*. We describe the details of those artifacts in Section 2.2.3.

2.2.3 Requirements Artifacts

The term “requirement artifact” refers to a documented requirement [Pohl, 2010]. As mentioned in Section 2.2.2, there are three main types of requirements artifacts: *goals*, *scenarios* and *solution-oriented requirements*.

2.2.3.1 Goals

Goals are used to document stakeholders’ intentions and refine the overall system objective into smaller objectives to be fulfilled by the subsystems. According to Pohl [Pohl, 2010], a goal is “an intention with regard to the objectives, properties, or use of the system”. van Lamsweerde [van Lamsweerde, 2001] defines a goal as “a high-level objective the system under consideration should achieve”.

There might be different goals at different levels of abstraction. High-level goals provide the high-level strategy for the product of the company, and the requirements engineer refines these high-level goals into sub-goals which define the stakeholders’ intention in terms of the use of the system and its specific properties [Pohl, 2010]. The refinement of a goal is called “goal-decomposition”. There are *AND-decomposition of a goal*, i.e., all sub-goals must be satisfied to satisfy the super goal, and *OR-decomposition of a goal*, i.e., satisfying one of the sub-goals is sufficient to satisfy the super goal. In addition to the AND-decomposition and OR-decomposition, there are dependencies between goals [Pohl, 2010]: *requires*, *support*, *obstruction*, *conflict* and *goal equivalence*.

Goals and goal dependencies can be documented using unstructured, natural language. For an easy-to-understand and precise documentation of goals, several goal modelling languages, methods and tools are proposed in literature (e.g., GRL [GRL, 2009], i* [Yu, 1997], NFR [Chung et al., 1996], GDC [Kavakli, 2002], GBRAM [Anton, 1996] and KAOS [van Lamsweerde, 2009]).

2.2.3.2 Scenarios

A scenario describes a concrete example of how the system to be developed interacts with its users and third-party systems. Pohl [Pohl, 2010] defines a scenario as “a concrete example of satisfying or failing to satisfy a goal (or set of goals)”. Scenarios describe sequences of actions related to the intended application. Scenarios can be documented using various formats, such as natural language, tabular notation, or sequence diagrams. According to Pohl and Haumer [Pohl and Haumer, 1997], there are three types of scenarios:

- **System Internal Scenarios:** They are used to represent interactions between components of the system and subsystems. They do not consider the context in which the system to be developed is expected to run.
- **Interaction Scenarios:** They are used to represent interactions between the system and stakeholders and/or other third-party systems.

- **Contextual Scenarios:** They extend interaction scenarios by additionally representing the system context information such as business goals related to the system services, relationships between stakeholders, and organisational policies.

Use case modeling is a technique to document interaction scenarios, and first introduced in OOSE (Object-Oriented Software Engineering) [Jacobson et al., 1992]. Pohl et al. [Pohl et al., 2005] define a use case as “a description of system behaviour in terms of scenarios illustrating different ways to succeed or fail in attaining one or more goals”.

A use case represents interactions between the actors and the system to be developed. These interactions are described in terms of scenarios, the so-called use case scenarios. There are usually various use case scenarios representing alternatives of failure and success for the same goal. Therefore, a use case consists of multiple positive and negative use case scenarios. Besides, use cases contain pre- and post-conditions providing information about the system state before and after the execution of the use case scenarios.

Use cases of a system are documented as a use case model in which three components are necessary [Larman, 2002]: the template-based specification of the use cases, the proper documentation of the use case scenarios, and at least one use case diagram representing with a graphical notation an overview of the use cases. A use case template is a tabular structure that guides the textual documentation of use cases [Pohl et al., 2005]. There are several use case templates in literature (e.g., [Cockburn, 2001] [Armour and Miller, 2001] [Kulak and Guiney, 2003]).

2.2.3.3 Solution-Oriented Requirements

Solution-oriented requirements specify, at the required level of detail, system properties and features [Pohl, 2010]. They describe the data, functional and behavioral perspectives on a software system. Different than goals and scenarios, they imply a conceptual solution for the system to be developed.

Conceptual or formal models are often used to document solution-oriented requirements. The requirements engineer can employ a data modelling language (e.g., the entity-relationship model [Chen, 1976], the enhanced entity relationship model [Elmasri and Navathe, 2015] and UML class diagrams) to document the solution-oriented requirements from the data perspective, while she employs a behavioral modelling language (e.g., data flow diagrams [DeMarco, 1978]) and a functional modelling language (e.g., finite automata, statecharts [Harel, 1987] and UML state machine diagrams) for the behavioral and functional perspectives, respectively. We consider the use of domain models in our Product line Use case modeling Method (PUM) (see Chapter 3) as the documentation of solution-oriented requirements in UML class diagrams from the data perspective.

2.3 Software Product Line Engineering

Software Product Line Engineering (SPLE) is described as "a reuse development paradigm that aims to develop a set of similar high-quality products at reasonable cost and within a short time to market" [Pohl et al., 2005]. It covers the development of pure software systems as well as the development of software that is embedded into a system integrating hardware and software [Pohl et al., 2005]. The goal of SPLE is to maximize reuse by creating a software product line from a shared set of software assets in a planned or predictive way. SPLE is currently becoming a common practice in industry to reduce cost and development cycle time and to improve software quality and productivity.

A software product line is defined as "a set of software-intensive systems sharing a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Clements and Northrop, 2001]. Software product line development refers to software engineering methods, tools and approaches for creating a set of similar software systems from a mutual set of software assets using a common means of production [CMU, 2018].

In this section, we present the core concepts of SPLE such as domain and application engineering activities and variability in product lines.

2.3.1 Software Product Line Framework

The principles of SPLE are represented in an SPLE framework (see Fig. 2.2). SPLE includes two processes: *domain engineering* and *application engineering*, where variability of the product line is defined and exploited, respectively (see e.g. [Pohl et al., 2005] [Weiss and Lai, 1999] [van der Linden, 2002]):

- **Domain engineering:** It is defined as "the process of software product line engineering in which the commonality and the variability of the product line are defined and realised" [Pohl et al., 2005]. All types of software artifacts are covered in this process (e.g., requirements specifications, design models, source code and test cases).
- **Application engineering:** It is defined as "the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability" [Pohl et al., 2005].

The advantage of having these two processes is the reduction of manual effort and time required to build customer-specific applications. To be efficient, these processes need to interact in a manner that is beneficial to both [Pohl et al., 2005]. We describe the details of the domain and application engineering processes in Sections 2.3.1.1 and 2.3.1.2, respectively.

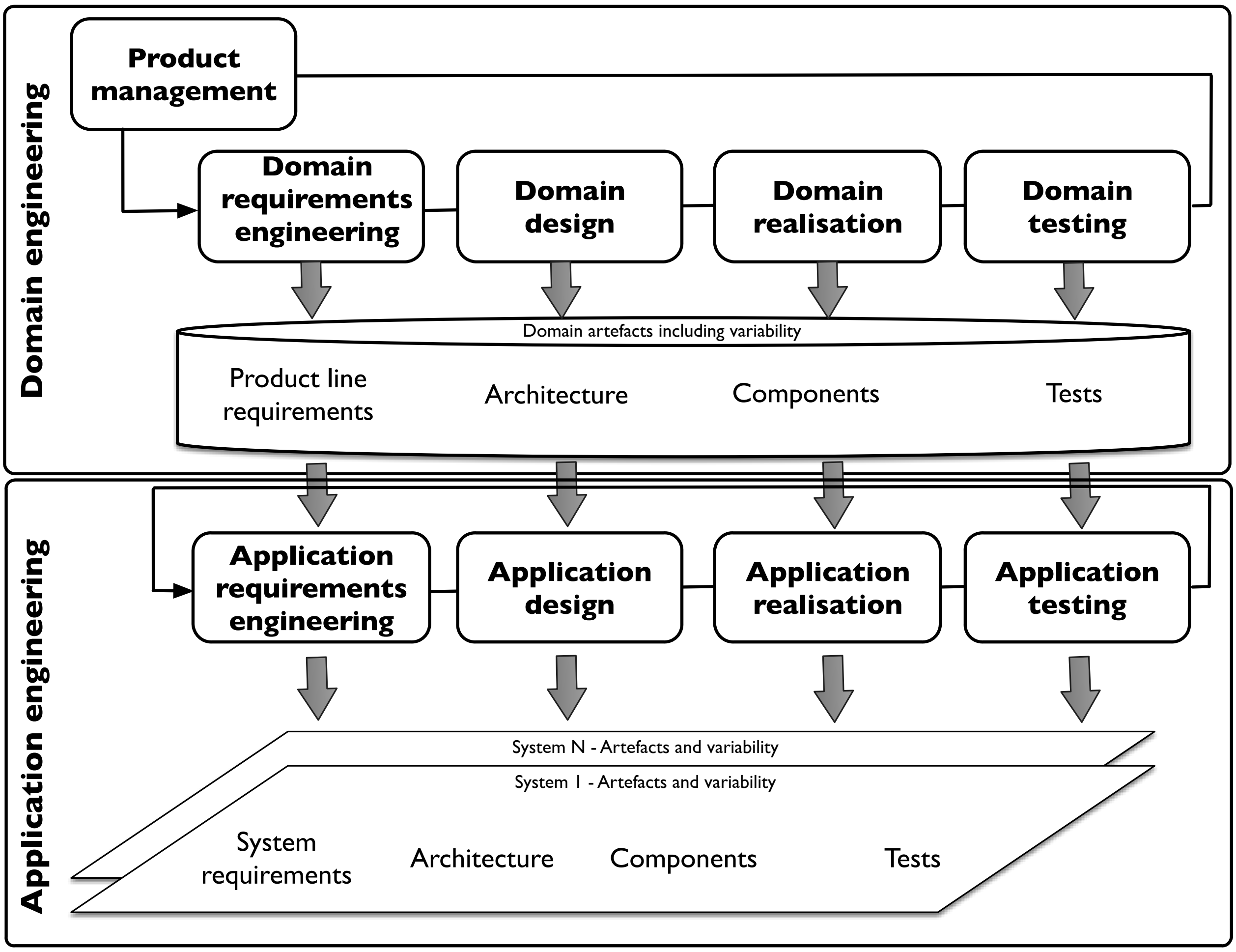


Figure 2.2. Software Product Line Engineering Framework [Pohl et al., 2005]

2.3.1.1 Domain Engineering

The objective of the domain engineering process is to develop reusable domain artifacts that define the commonality and the variability of the software product line and that can be reused for the development of the set of applications the software product line is planned for [Pohl et al., 2005].

The domain engineering process is composed of five key sub-processes: *product management*, *domain requirements engineering*, *domain design*, *domain realisation*, and *domain testing*. Below we briefly describe them.

- **Product management**. This concerns the economic aspects of the software product line such as the market strategy [Pohl et al., 2005]. The goal of this sub-process is to define the scope of the product line and to manage the products in the product line. Based on the objectives of the company owning the product line, in the product management sub-process, the engineer determines the schedule of the products and their planned release dates.
- **Domain requirements engineering**. This concerns the activities of eliciting and documenting common and variable requirements of the product line [Pohl et al., 2005]. The requirements of all the planned products in the software product line are defined in this sub-process. The

domain requirements engineering sub-process aims to reach an agreement among all the stakeholders on the requirements of the product line and to produce a precise documentation of the requirements.

- **Domain design**. This concerns the activities of the generation of the reference architecture of the product line [Pohl et al., 2005]. The reference architecture is a high-level structure for all the planned products in the software product line. In this sub-process, the engineer maps the product line requirements, including variability, to the reference architecture of the product line.
- **Domain realisation**. This concerns the detailed design and implementation of the reusable components and interfaces of the product line [Pohl et al., 2005]. In this sub-process, using the reference architecture, each component is planned, designed, and implemented to enable its reuse in application engineering.
- **Domain testing**. This concerns the validation and verification of the reusable components in the product line [Pohl et al., 2005]. In this sub-process, the components are tested against their requirements, architecture, and design. The objective of the domain testing sub-process is to validate the output of the previous domain engineering sub-processes and to improve the efficiency of testing by starting the testing process early in the domain engineering process.

2.3.1.2 Application Engineering

The objective of the application engineering process is to develop individual systems of the product line by reusing domain artifacts in the domain engineering process [Pohl et al., 2005]. The commonality and the variability of the software product line are exploited to achieve as high as possible reuse of the domain artifacts during the development of a product line application.

The application engineering process is composed of four key sub-processes: *application requirements engineering*, *application design*, *application realisation*, and *application testing*. Each sub-process uses domain artifacts to produce application artifacts (see Fig. 2.2). Below we briefly explain the application engineering sub-processes.

- **Application requirements engineering**. This concerns the activities of the definition and the documentation of application-specific requirements [Pohl et al., 2005]. The objective of this sub-process is to elicit and document requirements for a specific application and at the same time reuse, as much as possible, the domain requirements artifacts of the product line.
- **Application design**. This concerns the activities of the generation of the application architecture [Pohl et al., 2005]. This sub-process uses the reference architecture, generated in the domain design sub-process, to instantiate the application architecture. The resulting application architecture has to fulfill the application's requirements elicited and documented in the application engineering sub-process.
- **Application realisation**. This concerns the activities of the generation of the application-specific components and interfaces [Pohl et al., 2005]. This sub-process takes the application architecture as input and produces the executable application as an output.

- **Application testing**. This concerns the activities of the validation and verification of an application against its requirements [Pohl et al., 2005]. It focuses on the quality of the created application and aims to detect faults as early as possible.

2.3.2 Product Line Variability

The modeling of product line variability enables the development of applications by reusing predefined, adjustable artifacts [Pohl et al., 2005]. We refer to all the domain engineering activities concerned with the identification and documentation of variability as *modelling variability*. Variability modeling supports the development and reuse of variable development artifacts.

Variability is an essential property of domain artifacts in software product line engineering. The engineer introduces variability during the product management sub-process of domain engineering (see Fig. 2.2) when he identifies common and variable features of the applications in the software product line. Domain requirements engineering, domain design, domain realisation and domain testing deal with system artifacts at different levels of abstraction. At each level of abstraction in domain engineering, the engineer refines variability from the previous level and introduces more variability [Pohl et al., 2005].

Pohl et al. [Pohl et al., 2005] provide a variability metamodel which represents all the key concepts for variability (see Fig. 2.3). In the following, we give the definitions of these concepts.

Pohl et al. [Pohl et al., 2005] define a variation point as “a representation of a variable item of the real world or a variable property of such an item within domain artifacts enriched by contextual information”, and a variant as “a representation of a particular instance of a variable item or property within domain artifacts”. There are two types of variation points: *internal variation points* hidden from customers and *external variation points* visible to customers.

A variability dependency is a dependency between a variation point and its variant(s), which states that the variation point offers certain variant(s) [Pohl et al., 2005]. It may be one of two types: *mandatory* or *optional*. With an optional variability dependency, a variant can but does not have to be part of an application in the product line, whereas, with a mandatory variability dependency, a variant must be selected for an application when its variation point is part of the application [Pohl et al., 2005]. It is possible to define the minimum and the maximum number of optional variants to be selected from a set of variants of the variation point having an optional variability dependency.

There are also constraint dependencies between variants (i.e., variant constraint dependency), between variation points (i.e., variation point constraint dependency) and between variation points and variants (i.e., variation point to variant constraint dependency). A constraint dependency represents a restriction which is either of the type “requires” or “excludes”.

A variant constraint dependency is a relationship between two variants, which may be *variant requires variant* (i.e., requires_V_V) or *variant excludes variant* (i.e., excludes_V_V) [Pohl et al., 2005]. A variant to variation point constraint dependency is a relationship between a variant and

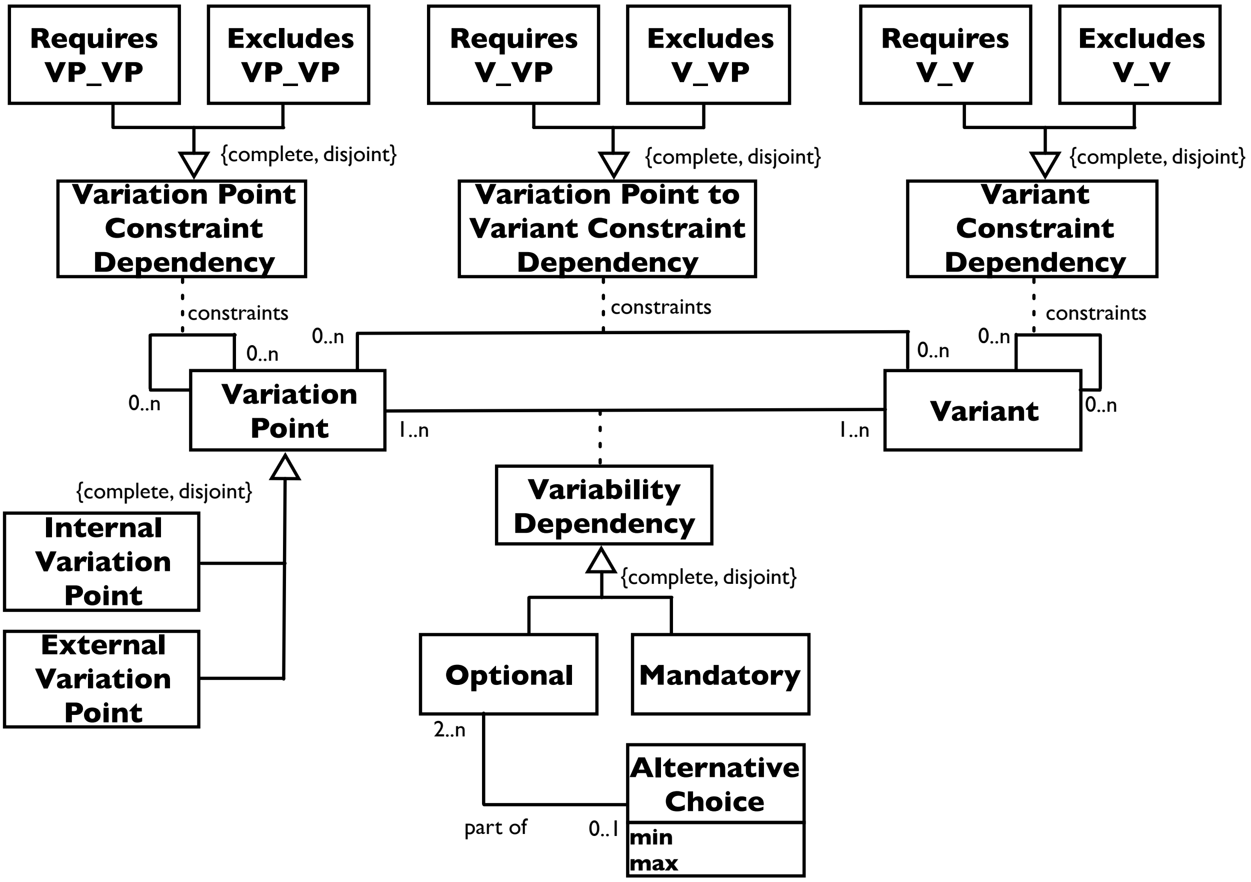


Figure 2.3. Variability Metamodel [Pohl et al., 2005]

a variation point, which may be *variant requires variation point* (i.e., requires_V_VP) or *variant excludes variation point* (i.e., excludes_V_VP) [Pohl et al., 2005]. A variation point constraint dependency describes a relationship between two variation points, which may be *variation point requires variation point* (i.e., requires_VP_VP) or *variation point excludes variation point* (i.e., excludes_VP_VP) [Pohl et al., 2005].

There are several languages and techniques to model variability in a product line. Feature modeling [Kang et al., 1990] has been the most popular technique to model variability in a product line. Commonalities and variabilities are modeled in terms of optional and mandatory product features, i.e., optional and mandatory characteristics of products in a product line visible to stakeholders [Kang and Lee, 2013]. The original feature model [Kang et al., 1990] is a simple model having features with "consists of" and "generalization/specialization" relationships in an AND/OR graph. Features can be mandatory, alternative, or optional. Many extensions have been proposed to add new concepts to the originally proposed feature model (e.g., [Czarnecki et al., 2004] [Czarnecki et al., 2002] [Eriksson et al., 2005a] [Kang et al., 1998] [Riebisch et al., 2002] [Gurp et al., 2001] [Benavides et al., 2005b] [Hein et al., 2000]). Orthogonal variability modeling is a more recent approach to document the variability of an SPL without taking into account common features [Pohl et al., 2005]. In an Orthogonal Variability Model (OVM), the variability of the product line is documented, while the OVM is externally traced to other software development artifacts such as textual requirements specifications, use case models, design models, source code, and test models.

2.4 Conclusion

In this chapter, we answered Research Question 1 (*What are the key concepts for managing requirements? What are the core concepts and techniques for managing variability in product families?*).

We introduced the basic concepts in requirements engineering and product line engineering. Our approach for supporting change in product lines within the context of use case-driven development and testing is based on some of the concepts described above, as pertaining to the objectives of this thesis.

Chapter 3

Product Line Use Case and Domain Models

In this chapter, we propose, apply, and assess Product line Use case modeling Method (PUM), an approach that supports modeling variability at different levels of granularity in use case and domain models. Our motivation is that, in many software development environments, use case modeling drives interactions among stakeholders and, therefore, use cases and domain models are common practice for requirements elicitation and analysis. In PUM, we integrate and adapt existing product line extensions for use case diagrams and introduce some template extensions for use case specifications. Variability is captured in use case diagrams while it is reflected at a greater level of detail in use case specifications. Variability in domain concepts is captured in domain models. PUM is supported by a tool relying on Natural Language Processing (NLP). We successfully applied PUM to an industrial automotive embedded system and report results from structured interviews with experienced engineers.

3.1 Introduction

In Chapter 1, we discussed in details that considerable research has been devoted to documenting variability in use cases, but many approaches [Eriksson et al., 2005a] [Eriksson et al., 2004] [Alferez et al., 2008] require that feature models be connected to use case specifications and diagrams. In such cases, feature modeling needs to be introduced into practice, including establishing and maintaining traces between feature models and use case specifications and diagrams, as well as other artifacts. In many development environments, such additional modeling and traceability is often perceived as an unnecessary overhead. In this chapter, we propose, apply, and assess a *Product line Use case modeling Method (PUM)*, which aims at enabling the analysts to document variability at different levels of granularity, both in use case diagrams and specifications, without requiring any feature model. In PUM, we adopt the product line extensions of use case diagrams proposed by Halmans and Pohl [Halmans and Pohl, 2003] to overcome the shortcomings associated with textual representation of variability. Further, we augment a more structured and analysable form of use case specifications, i.e., Restricted Use Case Modeling (RUCM) [Yue et al., 2013]. RUCM is based on a template and restriction rules, reducing the imprecision and incompleteness in use cases. We chose RUCM in PUM because it reduces ambiguity and facilitates automated analysis of use cases. RUCM was previously evaluated through controlled experiments and has shown to be usable and beneficial with respect to making use cases less ambigious and more amenable to precise analysis and design. Since RUCM was not originally designed for modeling variability in embedded systems, we introduce some extensions to represent the types of variability that cannot be captured in a use case diagram.

In addition to use case modeling, common practice in many environments also includes domain modeling, which aims at capturing domain concepts shared among stakeholders, with associated variability where needed. Such domain modeling is also part of PUM as it enables the use of consistent terminology and concepts, as well as precise definitions of conditions in use case specifications and, therefore, of related variation points. PUM expects these conditions, referring to the domain model, to be defined with the Object Constraint Language (OCL) [OCL, 2018] since OCL is the natural choice when defining high-level constraints on class diagrams. To summarize, the contributions of this chapter are:

- PUM, a use case modeling method, which integrates and builds on existing work and captures variability in product lines at a level of granularity enabling both precise communication and guided product configuration;
- a practical, industry-strength tool, relying on Natural Language Processing (NLP) to report inconsistencies between use case diagrams and use case specifications complying with the RUCM template;
- an industrial case study demonstrating the applicability of PUM, including structured interviews with experienced engineers.

In this chapter we answer *Research Question 2* raised in Chapter 1: *How to model variability in use case and domain models without additional traceability to feature models? Are there any existing techniques for use case and domain models to model variability? Which techniques can be used?* With the product line extensions employed in PUM, we address the need of modeling variability in Product Line (PL) use case and domain models without further requiring feature models.

In order to configure Product Specific (PS) use case and domain models, the configuration approach requires the analyst to make configuration decisions using the variability information in PL use case and domain models. The results in this chapter are used in Chapter 4 to automatically generate PS use case and domain models from PL use case and domain models, and configuration decisions.

This chapter is structured as follows. Section 3.2 introduces the industrial context of our case study to provide the motivations behind PUM. Section 3.3 discusses the related work. In Section 3.4, we provide an overview of PUM. Section 3.5 focuses on use case diagrams and RUCM with their extensions while Section 3.6 presents the tool support for PUM. In Section 3.7, we present our case study, i.e., Smart Trunk Opener (STO), along with interview results. We conclude the chapter in Section 3.8.

3.2 Motivation and Context

The context for which we developed PUM is that of automotive embedded systems, interacting with multiple external systems, and developed by a supplier for multiple automotive manufacturers. These systems are representative examples in which variability in requirements needs to be communicated among stakeholders, including customers. For instance, IEE negotiates, with each customer, how to resolve variation points, that is, the configuration of the product line.

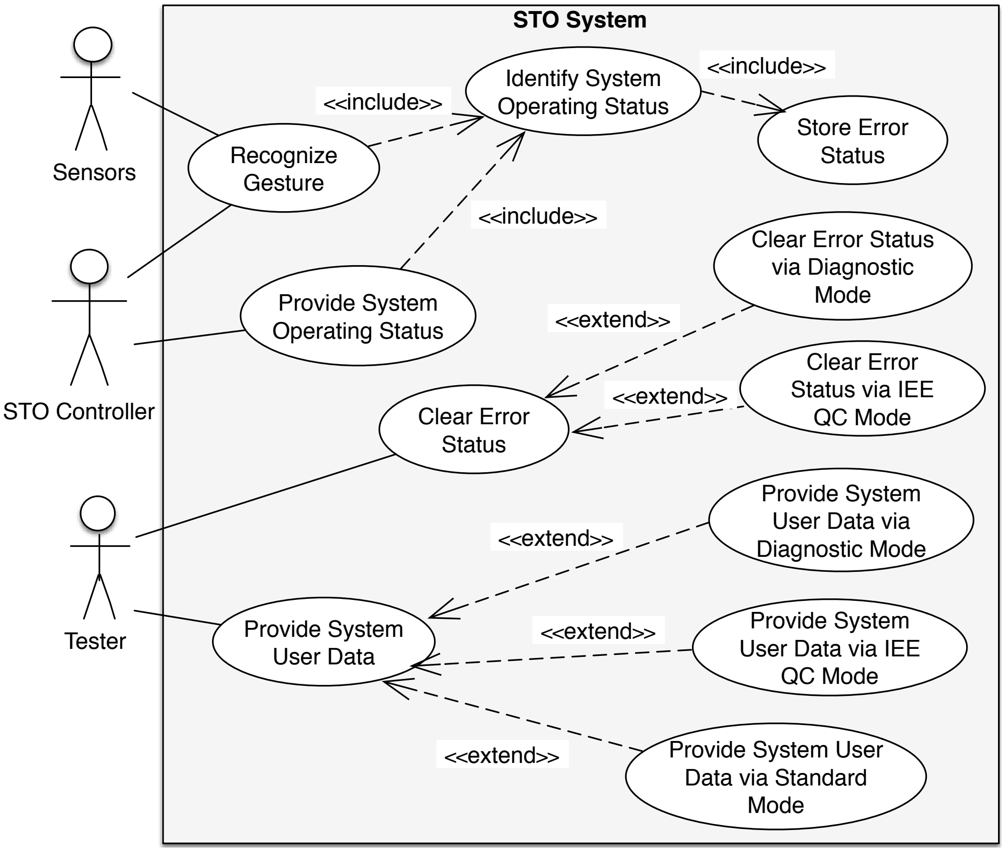

STO System
Sensors
STO Controller
Tester
Recognize Gesture
Identify System Operating Status
Store Error Status
Provide System Operating Status
Clear Error Status via Diagnostic Mode
Clear Error Status via IEE QC Mode
Clear Error Status
Provide System User Data via Diagnostic Mode
Provide System User Data via IEE QC Mode
Provide System User Data
Provide System User Data via Standard Mode
<<include>>
<<include>>
<<include>>
<<extend>>
<<extend>>
<<extend>>
<<extend>>
<<extend>>

Figure 3.1. Part of the UML Use Case Diagram for STO

The current use case driven development practice at IEE involves UML use case diagrams and use case specifications. Fig. 3.1 depicts part of the UML use case diagram for STO. *Sensors*, *STO Controller* and *Tester* are the actors of the system. The use cases describe four main functions: *recognize gesture*, *provide system operating status*, *provide system user data*, and *clear errors stored in the system*.

UML provides the *extend* and *include* relations in use case diagrams to extend the behaviour of use cases and to factor out common parts of the behaviors of two or more use cases, respectively. However, it is not possible to explicitly document variability, e.g., which use cases are mandatory and which use cases are variant. There is no way to represent variation points, i.e., location at which a variation occurs. We know from our discussions with IEE that, in Fig. 3.1, there are three variation points: *Store Error Status*, *Clear Error Status* and *Provide System User Data*. *Store Error Status* and *Clear Error Status* are optional while *Provide System User Data* is mandatory. Cardinality in a variation point, i.e., number of variants to be chosen, cannot be represented in a UML use case diagram [Halmans and Pohl, 2003]. For instance, there must be at least two ways to realize *Provide System User Data* (i.e., at least two use cases that extend the 'Provide System User Data' use case) in any STO product. Variant dependencies play a key role in the selection of variants. For example, if an STO product provides *Store Error Status*, then it must also provide *Clear Error Status*.

A use case specification contains detailed description of a use case given in a use case diagram and usually conforms to a use case template [Cockburn, 2001] [Armour and Miller, 2001] [Kulak and Guiney, 2003]. IEE has so far been following the Cockburn's template [Cockburn, 2001] (see Table 3.1 for some simplified STO use cases).

Standard use case templates, such as Cockburn's, are insufficient to document variability in use case specifications. For example, variation points and variants are not visible in the STO use case specifications. Variant use cases, e.g., *Provide System User Data via Standard Mode* in Lines 21-28, are not distinguishable from any other use case. In Lines 18-19 and 25-28 there are two variation points modelled as extensions of the basic flow, i.e., executing the variant use cases *Store Error Status*, *Provide System User Data via Standard Mode, IEE QC Mode, and Diagnostic Mode*. However, these steps are no different from any other step describing the execution of a mandatory use case, e.g., the execution of the *Identify System Operating Status* use case in Line 2. Some use case steps might be optional or their order may vary in the product line. For instance, all steps given in Lines 32-36 are optional with a variant order. The analyst first needs to properly document the optional steps. Then, she can negotiate with the customer to decide which steps to select, according to which order, in the product.

Within the context of developing industrial automotive embedded systems for multiple manufacturers, we identify three challenges that need to be considered in capturing requirements' variability in use cases:

Challenge 1: **Modeling Variability with Constraints and Dependencies.** It is crucial to have variability information explicitly documented (i.e., variants, variation points, their constraints and dependencies) in order to decide with the customers which variants to include for the product and to guide product configuration. Textual representation of use cases has shortcomings in explicitly representing variability information. Furthermore, it is not easy for analysts and customers to comprehend and visualize all variability information encoded in a textual representation. For instance, in STO, we identified 15 mandatory and 14 variant use cases which contain 8 variation points, and 7 variant dependencies. The STO use cases include 244 use case flows (29 basic flows, 202 alternative flows, and 13 optional alternative flows). The variability information scattered across all these use case flows in the textual description needs to be communicated to customers and used to configure a product.

Challenge 2: **Reflecting Variability in Use Case Specifications.** There are approaches that extend only use case diagrams with the notion of variation point and variant to express variability and associated constraints. IEE, as well as many similar companies, rely on detailed use case specifications to communicate with their customers. Variability should therefore be reflected in use case

Table 3.1. Some Use Cases for STO

1	**USE CASE** Recognize Gesture
2	1. The system *'identifies system operating status'*.
3	2. The system receives the move capacitance from the sensors.
4	3. The system confirms the movement is a valid kick.
5	4. The system informs the trunk controller about the valid kick.
6	**Extensions**
7	3a. The movement is not a valid kick.
8	3a1. The system sets the overuse counter.
9	
10	**USE CASE** Identify System Operating Status
11	**Main Success Scenario**
12	1. The system checks Watchdog reset and RAM.
13	2. The system checks the upper and lower sensors.
14	3. The system checks if there is any error detected.
15	**Extensions**
16	2a. Sensors are not working properly.
17	2a1. The system identifies a sensor error.
18	3a. There is an error in the system.
19	3a1. The system stores error status.
20	
21	**USE CASE** Provide System User Data
22	1. The tester requests receiving system user data via standard mode.
23	2. The system *'provides system user data via Standard Mode'*.
24	**Extensions**
25	1a. The tester requests receiving user data via diagnostic mode.
26	1a1. The system *'provides system user data via Diagnostic Mode'*.
27	1b. The tester requests receiving system user data via IEE QC mode.
28	1b1. The system *'provides system user data via IEE QC Mode'*.
29	
30	**USE CASE** Provide System User Data via Standard Mode
31	**Main Success Scenario**
32	1. The system sends the calibration data to the tester.
33	2. The system sends the trace data to the tester.
34	3. The system sends the error data to the tester.
35	4. The system sends the sensor data to the tester.
36	5. The system sends the error trace data to the tester.

specifications to provide diagram-specification consistency. In addition, there are types of variability (e.g., optional steps) which cannot be expressed in use case diagrams, at the required level of granularity, to precisely guide product configuration. However, it is nevertheless crucial to also model some variability information in use case diagrams, for example to improve visualization of key variability information and to provide a road map to look up the details in use case specifications.

Challenge 3: **Capturing Variability with Precise Conditions.** Use cases are not meant to clarify terminology and domain concepts shared among all stakeholders. In order to document domain concepts and their variability, any product line requirements modeling method would need models where the analyst can specify mandatory and optional domain entities. In addition, 'flows of events' in use cases feature associated conditions determining their occurrence, which need to be precisely specified to help communication among stakeholders. For instance, the precise definition of a valid kick (see Lines 4 and 7 in Table 3.1) is crucial for the IEE engineers to identify the correct execution of the product in terms of 'flows of events' in the STO use cases.

In the remainder of this chapter, we focus on how to best address these three challenges in a practical manner, in the context of use case driven development, while minimising the modeling overhead. Automated configuration, change impact analysis and regression testing are three potential applications of product line use case modeling, which we provide in Chapters 4, 5 and 6. In collaboration with customers, analysts need to discuss variability documented in use cases to configure the required parts of the system design, implementation, and test cases. Customers may change their decisions during or after configuration as the product evolves. Therefore, in addition to configuration, analysts need to perform change impact analysis to identify what other decisions may be impacted and thus what artifacts must be updated.

3.3 Related Work

In this section, we cover related work across four categories.

Relating Feature Models and Use Cases. Some approaches propose using feature models for modeling variability information within the context of use case driven development (*Challenges 1 and 2*). Griss et al. [Griss et al., 1998] describe a manual process for constructing a feature model from a use case diagram. The main idea is to extract the structure of feature models from use case dependencies (i.e., *include* and *extend*) in use case diagrams. Braganca et al. [Braganca and Machado, 2007] investigate the use of model transformation to automate the same idea but their approach requires that each feature be mapped to only one use case. Eriksson et al. [Eriksson et al., 2005a] [Eriksson et al., 2005b] [Eriksson et al., 2009] propose another approach relating use cases and features at a lower level of granularity, i.e., sequences of use case steps. Buhne et al. [Buhne et al., 2006] use Orthogonal Variability Models (OVMs) traced to use case diagrams and specifications. An OVM documents the variable aspects of a product line by using variation points, variants and their dependencies. Following traces, analysts can identify how a given variant in the OVM is implemented in use case diagrams and specifications. Alferez et al. [Alferez et al., 2008] provide a trace metamodel to capture traces between feature models and use cases. XTraQue [Jirapanthong and Zisman, 2009] is a tool which supports semi-automatic trace generation for use cases and feature models. Despite advances in traceability research, all approaches given above bring additional modeling and traceability effort into practice. Furthermore, their correctness highly depends on the correctness and precision of traces. To use these approaches, in most cases, traces between variability models and other modeling artifacts, e.g., use cases, need to be manually established at a very low level of granularity, e.g., conditions in use case steps. Traces should be maintained for every single change in any traced artifact. Our objective is to achieve the same result by solely relying on use case and domain modeling, which are common practice.

Extending Use Case Templates. Some works propose use case templates with product line extensions to model variability in use case specifications (*Challenges 1 and 2*). Gallina and Guelfi [Gallina and Guelfi, 2007] provide a product line use case template in which variants and variation points can be expressed. The template requires that variability information be encoded in use case specifications containing the fields 'selection category' and 'variation point'. These two fields do not follow any structured format to precisely define variability, in order, for example, to support product configuration. The representation of variation point cardinalities is not addressed in the template. Biddle et al. [Biddle et al., 2002] provide support for customizing use cases through parametrization. Nebut et al. [Nebut et al., 2006] enhance a use case template with parameters and contracts for product line system testing. These two approaches using parameters do not allow analysts to explicitly document variants and variation points. Fantechi et al. [Fantechi et al., 2004b] [Fantechi et al., 2004a] propose Product Line Use Cases (PLUCs), an extension of the Cockburn's use case template with three kinds of tags (i.e., *alternative*, *parametric*, and *optional*). It is not possible with these tags to explicitly represent mandatory and optional variants. Variants and variation points are hidden in use case specifications conforming to PLUC. In most of the approaches given above, either variants and variation points cannot be documented or it is not possible to express all required types of variability constraints. It is crucial to explicitly document variability information containing variants and variation points with all their constraints and dependencies (*Challenge 1*) since analysts and customers need them to make decisions during configuration.

Extending Use Case Diagrams. Variability modeling in use cases are also addressed by approaches extending use case diagrams with new relations and stereotypes (*Challenge 1*). Maßen and Lichter [von der Maßen and Lichter, 2002] propose two new relations to represent alternative and optional use cases in UML use case diagrams, without any support for expressing variation points. Azevedo et al. [Azevedo et al., 2012] [Azevedo et al., 2010] explore the use of the UML 'extend' relation with the new stereotypes 'alternative', 'specialization' and 'option' to distinguish variability types. The 'alternative' and 'specialization' are applied to the 'extend' relation while the 'option' is applied to use cases that represent options. The use of the 'extend' with the stereotypes does not address variation points and their cardinalities. John and Muthig [John and Muthig, 2004] introduce the stereotype 'variant' to use case diagrams. They also use the tag 'variant' for variant text fragments in use case specifications. They propose a new artifact, called decision model, to represent variation points textually but such a decision model can quickly become too complex for the analyst to comprehend. Halmans and Pohl [Halmans and Pohl, 2003] propose extensions to use case diagrams to explicitly represent variants, variation points, and associated constraints. Buhne et al. [Buhne et al., 2003] enhance the extensions with some common dependency types from feature modeling. Based on our observations in practice, Halmans et al.'s extensions support a subset of our needs (*Challenge 1*) and we therefore include them in our methodology. The approaches above do not reflect variability in use case specifications (*Challenge 2*) since they do not use any template extensions. Therefore, in PUM, we introduce extensions into RUCM to reflect variability in use case specifications and also to represent the types of variability, e.g., optional use case steps, that cannot be captured in use case diagrams (*Challenge 2*).

Capturing Variability in Domain Models. Variability in domain models is mostly addressed by introducing new stereotypes into UML class diagrams. Ziadi and Jezequel [Ziadi and Jezequel, 2006] suggest three stereotypes (i.e., *variant*, *variation*, and *optional*) to specify variability in domain models. These stereotypes are very similar to the 'kernel' and 'optional' stereotypes proposed by Gomaa [Gomaa, 2000]. In addition, Ziadi and Jezequel suggest using OCL to specify dependencies between variants in domain models, e.g., the presence of a variant requires the presence of another variant. In contrast, in PUM, we specify these dependencies in use case diagrams. We employ OCL and domain models to precisely specify conditions associated with flows of events in use case specifications (*Challenge 3*).

3.4 Overview of Our Modeling Method

As depicted in Fig. 3.2, PUM is designed to address the challenges stated above in the use case driven development context we described, and builds upon and integrates existing work. The PUM output is a *product line use case diagram*, *product line use case specifications*, a *domain model*, and *OCL constraints*. Variability, and its constraints and dependencies, are captured in the use case diagram (*Challenge 1*) while it is further detailed in the use case specifications (*Challenge 2*). Variability in domain concepts is captured in the domain model (*Challenge 3*). Use case conditions are reformulated as OCL constraints referring to the domain model (*Challenge 3*).

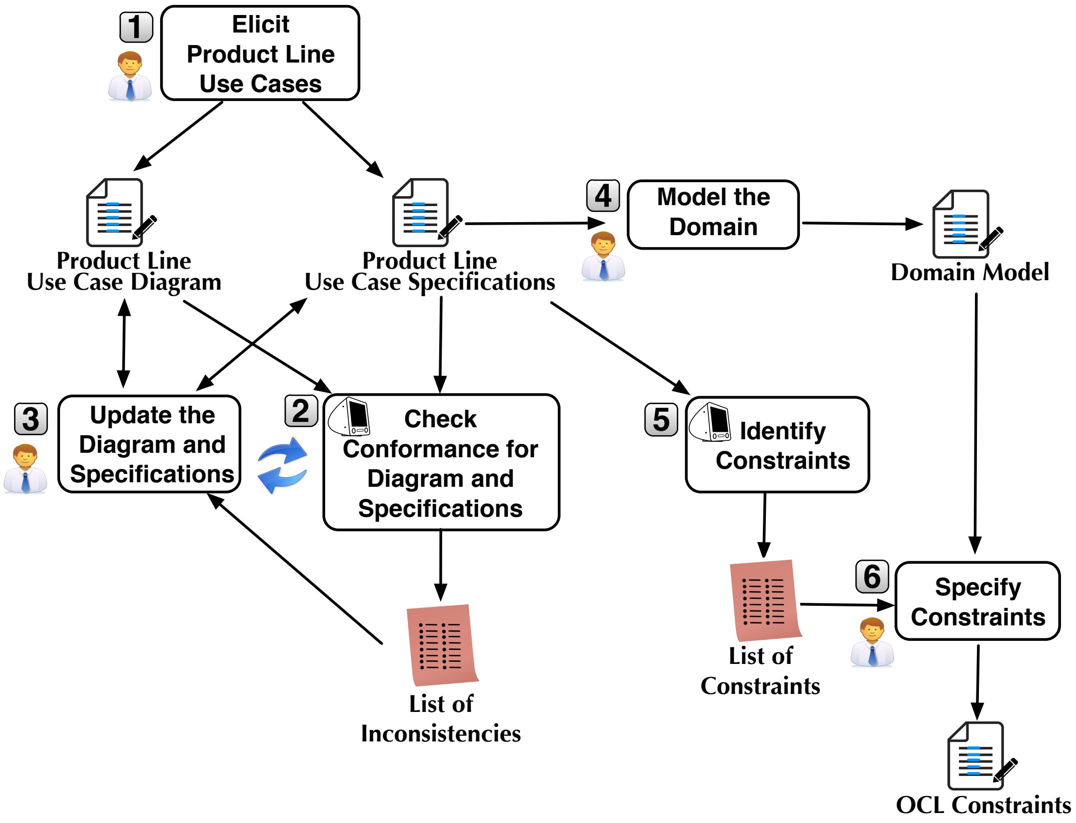

1
Elicit Product Line Use Cases
Product Line Use Case Diagram
Product Line Use Case Specifications
4
Model the Domain
Domain Model
3
Update the Diagram and Specifications
2
Check Conformance for Diagram and Specifications
5
Identify Constraints
List of Inconsistencies
List of Constraints
6
Specify Constraints
OCL Constraints

Figure 3.2. Overview of PUM

The analyst elicits product line use cases with the use case diagram, the RUCM template, and their product line extensions (Step 1). PUM-C (Product line Use case Modeling - Checker), the tool we developed for PUM, automatically checks for use case diagram and specification consistency (RUCM template) and reports inconsistencies (Step 2). The tool relies on Natural Language Processing (NLP). If there is any inconsistency, the analyst updates the diagram and/or specifications (Step 3). Steps 2 and 3 are iterative: the specifications and diagram are updated until the specifications conform to the RUCM template and they are consistent with the diagram.

The domain model is manually created as a UML class diagram by the analyst (Step 4). It is important for the analyst to clarify domain concepts shared among stakeholders. Variability in these concepts is expressed in the domain model by tagging domain entities as *variation*, *variant*, and *optional*. After the model is completed, textual descriptions of conditions in the use case specifications are automatically extracted (Step 5) to be reformulated from English to OCL by the analyst (Step 6).

The rest of the chapter provides a detailed description of each step in PUM, along with detailed illustrations from STO.

3.5 Capturing Variability in Requirements

In this section, we provide a detailed description of the artifacts produced by PUM. We also highlight how they were extended, compared to what was proposed in existing work, to address our needs.

3.5.1 Use Case Diagram with Product Line Extensions

PUM uses the product line extensions of use case diagrams proposed by Halmans and Pohl [Halmans and Pohl, 2003]. We do not introduce any further additions into the extensions. We chose these extensions for PUM because they support explicit representation of variants, variation points, and their depedencies (*Challenge 1*). In this section, we briefly define the extensions and the reader is referred to [Halmans and Pohl, 2003] [Buhne et al., 2003] for further details. Fig. 3.3 depicts the graphical notation of the extensions.

Variant use cases are distinguished from essential use cases, i.e., mandatory for all products in a product family, by using the ‘Variant’ stereotype (Fig. 3.3(a)). A variation point given as a triangle is associated to one, or more than one use case using the ‘include’ relation. A ‘tree-like’ relation, containing a cardinality constraint, is used to express relations between variants and variation points, which are called *variability relations*. The relation uses a [min..max] notation in which *min* and *max* define the minimum and maximum numbers of variants that can be selected for the variation point. A variability relation is optional where ($min = 0$) or ($min > 0$ and $max < n$); n is the number of variants for a variation point. A relation is mandatory where ($min = max = n$). The customer has no choice when a mandatory relation relates mandatory variants to a variation point [Halmans and Pohl, 2003]. Optional and mandatory relations are depicted with light-grey and black filled circles, respectively (Fig. 3.3(b)).

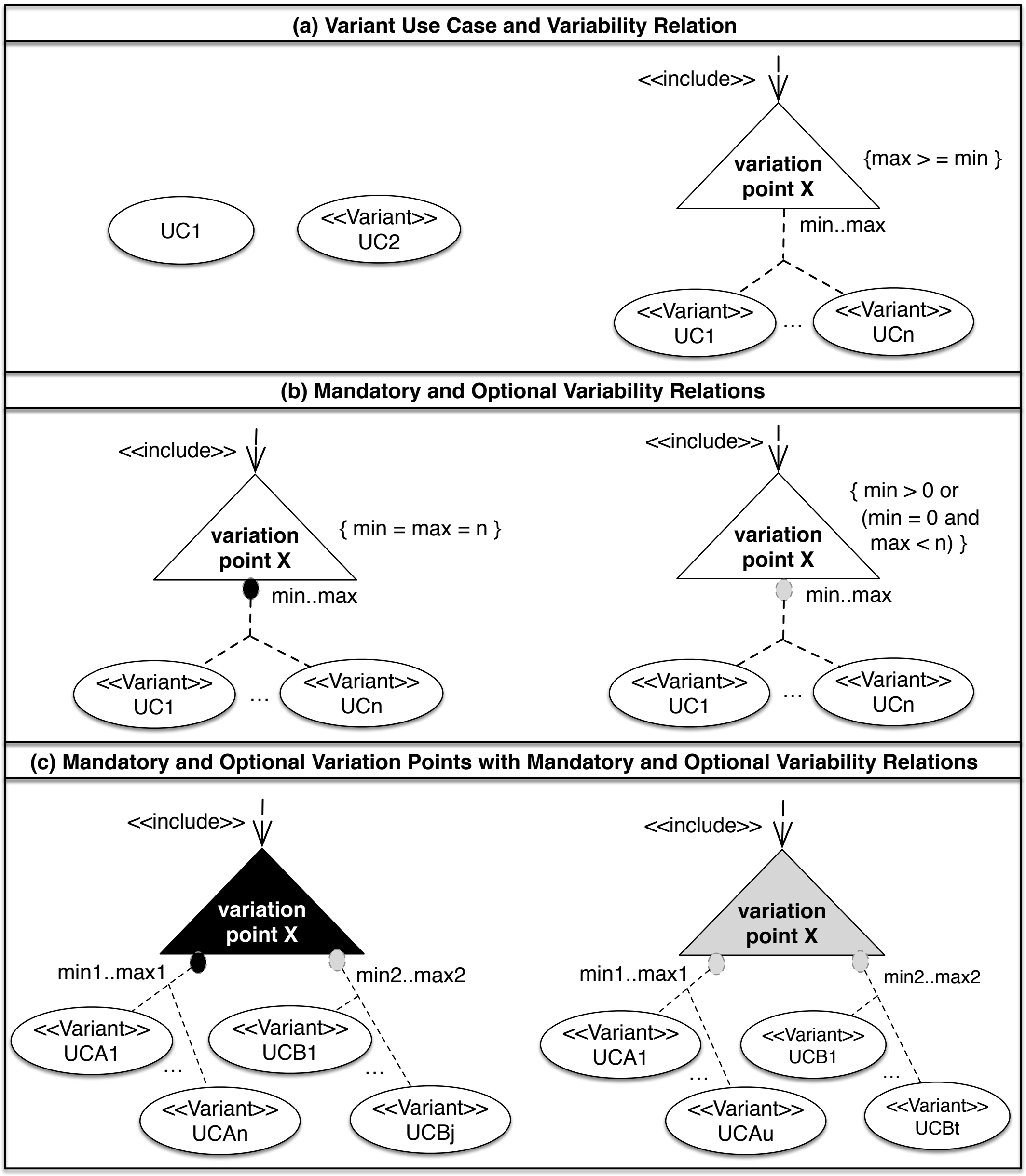


Figure 3.3. Graphical Notation of Product Line Extensions for Use Case Diagrams

Multiple variability relations can be combined to specify the desired cardinality in a variation point [Halmans and Pohl, 2003]. A variation point is optional or mandatory based on its variability relations. A variation point is *mandatory* if ($min > 0$) in at least one variability relation for that variation point. It is *optional* if ($min = 0$) in all its variability relations. Optional and mandatory variation points are rendered as grey and black-filled triangles, respectively (Fig. 3.3(c)). Besides the ‘include’ relation, two more variant relations for variants and variation points (i.e., ‘require’ and ‘exclusive’ [Buhne et al., 2003]) are used in PUM since these two relations model the consequences of decisions in product configuration. For instance, a variant might require or exclude the choice of another variant. Halmans and Pohl do not provide any metamodel or UML profile for the extensions in their paper [Halmans and Pohl, 2003]. In Fig. 3.4, to facilitate the tailoring of use case modeling tools, such a metamodel is depicted.

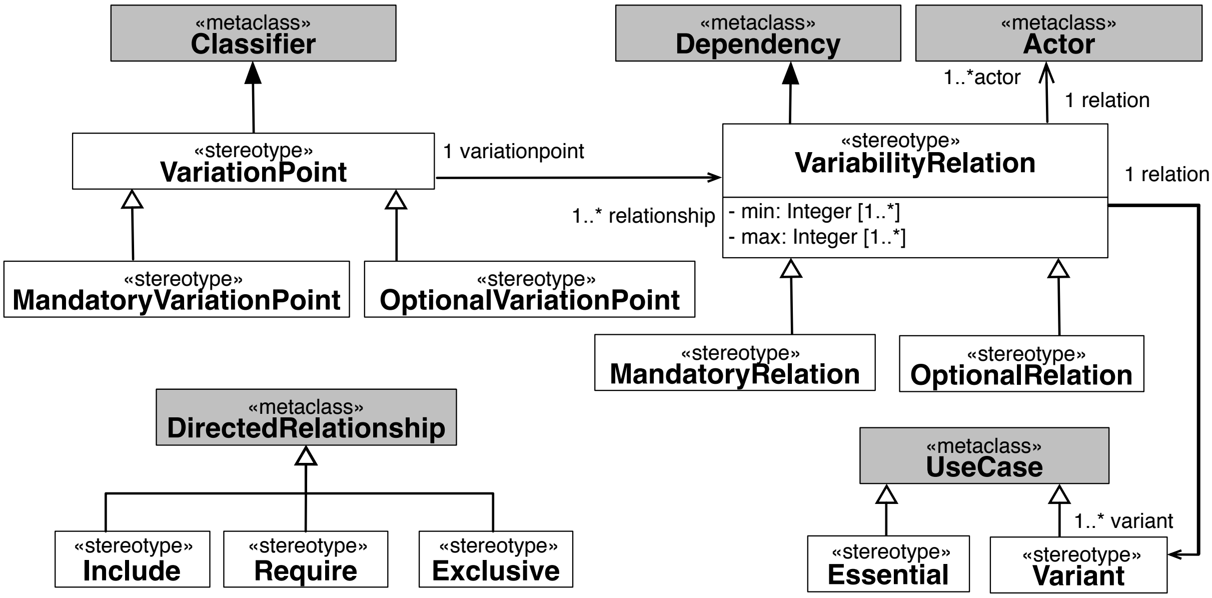


Figure 3.4. Extended UML Use Case Metamodel

A use case is extended as *Essential* and *Variant*. The notation for variation points, in Fig. 3.3, can also be applied to specify variation points for actors [Halmans and Pohl, 2003]. Therefore, a variability relation relates a variation point either to a variant use case or to an actor (see *VariabilityRelation* in Fig. 3.4).

Fig. 3.5 gives part of the product line use case diagram for STO. We document two optional and two mandatory variation points. The mandatory variation points indicate where the customer has to make a selection for an STO product. For instance, the 'Provide System User Data' essential use case has to support multiple methods of providing data where the methods of providing data via IEE QC mode and Standard mode are mandatory (the mandatory variability relation in the 'Method of Providing Data' variation point with a cardinality of '2 ..2'). In addition, the customer can select the method of sending data via diagnostic mode, i.e., the 'Provide System User Data via Diagnostic Mode' variant use case with an optional variability relation. In STO, the customer may decide that the system does not store the errors determined while the system identifies its operating status (the 'Identify System Operating Status' essential use case and the 'Storing Error Status' optional variation point). The 'require' relation relates the two optional variation points such that if the customer selects the variant use case in the 'Storing Error Status' variation point, he has to select the variant use case in the 'Clearing Error Status' variation point.

In use case diagrams, we capture variants, variation points, their cardinalities and dependencies. However, some detailed information of variability cannot be captured in these diagrams. For instance, the diagram in Fig. 3.5 indicates that the 'Identify System Operating Status' use case includes the 'Storing Error Status' optional variation point. To find out in which flows of events the variation point is included, the analyst has to check the corresponding use case specification.

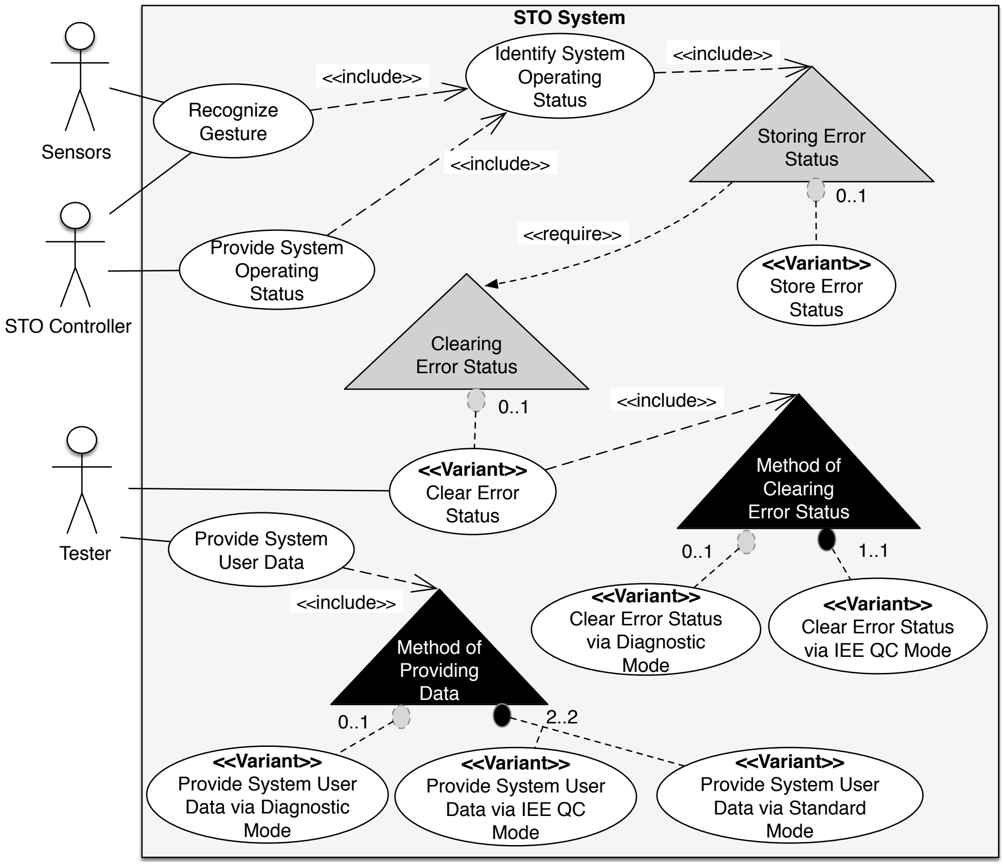

STO System
Sensors
STO Controller
Tester
Recognize Gesture
<<include>>
Identify System Operating Status
<<include>>
Storing Error Status
0..1
<<Variant>> Store Error Status
<<include>>
Provide System Operating Status
<<require>>
Clearing Error Status
0..1
<<include>>
<<Variant>> Clear Error Status
Method of Clearing Error Status
0..1
1..1
<<Variant>> Clear Error Status via Diagnostic Mode
<<Variant>> Clear Error Status via IEE QC Mode
Provide System User Data
<<include>>
Method of Providing Data
0..1
2..2
<<Variant>> Provide System User Data via Diagnostic Mode
<<Variant>> Provide System User Data via IEE QC Mode
<<Variant>> Provide System User Data via Standard Mode

Figure 3.5. Part of the Product Line Use Case Diagram for STO

3.5.2 Restricted Use Case Modeling (RUCM) and its Extensions

This section briefly introduces the RUCM template and our extensions for variability modeling in embedded systems. RUCM provides restriction rules and specific keywords constraining the use of natural language in use case specifications [Yue et al., 2013]. We chose RUCM for PUM since it was designed to make use case specifications more precise and analyzable, while preserving their readability. But since it was not originally designed for product line modeling of embedded systems, we had to introduce extensions (*Challenge 2*).

Table 3.2 provides some STO use cases written according to the extended RUCM rules. In RUCM, use cases have basic and alternative flows (Lines 2, 8, 13, 16, 22, 27, 33 and 38). In Table 3.2, we omit some alternative flows and some basic information such as actors and pre/post conditions.

A basic flow describes a main successful path that satisfies stakeholder interests. It contains use case steps and a postcondition (Lines 3-7, 23-26 and 39-43). A step can be one of the following interactions: an actor sends a request and/or data to the system (Lines 34); the system validates a request and/or data (Line 4); the system replies to an actor with a result (Line 7). A step can also capture the system altering its internal state (Line 18). In addition, the inclusion of another use case is specified as a step. This is the case of Line 4, as denoted by the keyword '*INCLUDE USE CASE*'. All keywords are written in capital letters for readability.

Table 3.2. Some STO Use Cases in the extended RUCM

1	**USE CASE** Recognize Gesture
2	**1.1 Basic Flow**
3	1. The system REQUESTS the move capacitance FROM the sensors.
4	2. INCLUDE USE CASE Identify System Operating Status.
5	3. The system VALIDATES THAT the operating status is valid.
6	4. The system VALIDATES THAT the movement is a valid kick.
7	5. The system SENDS the valid kick status TO the STO Controller.
8	**1.2 <OPTIONAL>Bounded Alternative Flow**
9	RFS 1-4
10	1. IF voltage fluctuation is detected THEN
11	2. RESUME STEP 1.
12	3. ENDIF
13	**1.3 Specific Alternative Flow**
14	RFS 3
15	1. ABORT.
16	**1.4 Specific Alternative Flow**
17	RFS 4
18	1. The system increments the OveruseCounter by the increment step.
19	2. ABORT.
20	
21	**USE CASE** Identify System Operating Status
22	**1.1 Basic Flow**
23	1. The system VALIDATES THAT the watchdog reset is valid.
24	2. The system VALIDATES THAT the RAM is valid.
25	3. The system VALIDATES THAT the sensors are valid.
26	4. The system VALIDATES THAT there is no error detected.
27	**1.4 Specific Alternative Flow**
28	RFS 4
29	1. INCLUDE <VARIATION POINT: Storing Error Status>.
30	2. ABORT.
31	
32	**USE CASE** Provide System User Data
33	**1.1 Basic Flow**
34	1. The tester SENDS the system user data request TO the system.
35	2. INCLUDE <VARIATION POINT : Method of Providing Data>.
36	
37	**<VARIANT>USE CASE** Provide System User Data via Standard Mode
38	**1.1 Basic Flow**
39	V1. <OPTIONAL>The system SENDS calibration TO the tester.
40	V2. <OPTIONAL>The system SENDS sensor data TO the tester.
41	V3. <OPTIONAL>The system SENDS trace data TO the tester.
42	V4. <OPTIONAL>The system SENDS error data TO the tester.
43	V5. <OPTIONAL>The system SENDS error trace data TO the tester.

The keyword '*VALIDATES THAT*' (Line 5) indicates a condition that must be true to take the next step, otherwise an alternative flow is taken. In Table 3.2, the system proceeds to Step 4 (Line 6) if the operating status is valid (Line 5).

Alternative flows describe other scenarios, both success and failure. An alternative flow always depends on a condition in a specific step of the basic flow. In RUCM, there are three types of alternative flows: *specific*, *bounded* and *global*. A specific alternative flow refers to a step in the basic flow (Lines 13, 16 and 27). A bounded alternative flow refers to more than one step in the basic flow (Line 8) while a global alternative flow refers to any step in the basic flow. For specific and bounded alternative flows, the keyword '*RFS*' is used to refer to one or more reference flow steps (Lines 9, 14, 17, and 28).

Bounded and global alternative flows begin with the keyword '*IF .. THEN*' for the condition under which the alternative flow is taken (Line 10). Specific alternative flows do not necessarily begin with '*IF .. THEN*' since a guard condition is already indicated in its reference flow step (Line 5).

Our RUCM extensions are twofold: (i) new keywords and restriction rules for modeling interactions in embedded systems and restricting the use of existing keywords; (ii) new keywords for modeling variability in use case specifications.

PUM introduces extensions into RUCM regarding the usage of '*IF*' conditions and the way input/output messages are expressed. PUM follows the guidelines that suggest not to use multiple branches within the same use case path [Larman, 2002], thus enforcing the usage of '*IF*' conditions only as a means to specify guard conditions for alternative flows. PUM introduces the keywords '*SENDS .. TO*' and '*REQUESTS .. FROM*' to distinguish system-actor interactions. According to our experience, in embedded systems, system-actor interactions are always specified in terms of messages. For instance, Step 1 in Table 3.2 (Line 3) indicates an input message from the sensors to the system while Step 5 (Line 7) contains an output message from the system to the STO Controller. Additional keywords can be defined for other types of systems.

To reflect variability in use case specifications in a restricted form, we introduce the notion of variation point and variant, complementary to the diagram extensions in Section 3.5.1, into the RUCM template. Variation points can be included in basic or alternative flows of use cases. We employ the '*INCLUDE <VARIATION POINT : ... >*' keyword to specify the inclusion of variation points in use case specifications (Lines 29 and 35). Variant use cases are given with the '*<VARIANT >*' keyword (Line 37). The same keyword is also used for variant actors related to a variation point given in the use case diagram.

There are types of variability (e.g, optional steps and optional alternative flows) which cannot be captured in use case diagrams due to the required level of granularity for product configuration. To model such variability, as part of the RUCM template extensions, we introduce optional steps, optional alternative flows and variant order of steps. Optional steps and optional alternative flows begin with the '*<OPTIONAL>*' keyword (Lines 8 and 39-43). In addition, the order of use case steps may also vary. We use the 'V' keyword before the step number to express the variant step order (Lines 39-43). Variant order occurs with optional and/or mandatory steps. It is important because variability in the system behavior can be introduced by multiple execution orders of the same steps. For instance, the steps of the basic flow of the 'Provide System User Data via Standard Mode' use case are optional. Based on the testing procedure followed in the STO product, the order of sending data to the tester also varies. In the product configuration, the customer has to decide which optional step to include in which order in the use case specification.

3.5.3 Domain Model and OCL Constraints

PUM uses the stereotypes (i.e., *variation*, *variant*, and *optional*) provided by Ziadi and Jezequel [Ziadi and Jezequel, 2006] to model variability with domain models (*Challenge 3*) since they support two common mechanisms to specify variability in UML class diagrams, i.e., *optionality* and *variation* (see Fig. 3.6).

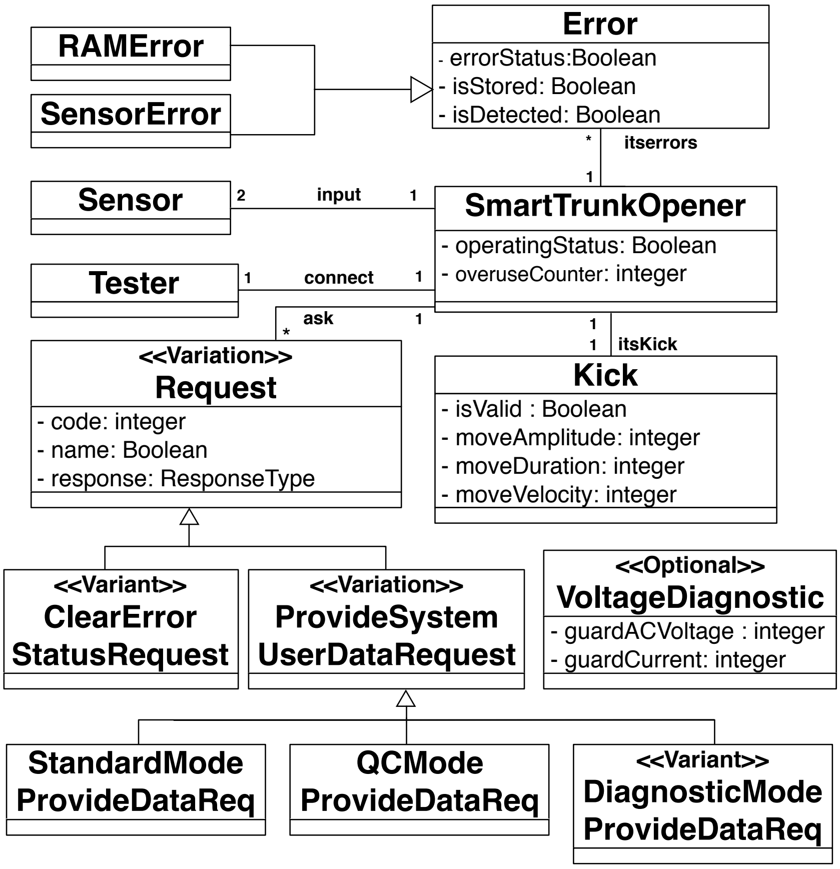


Figure 3.6. Simplified Portion of the Domain Model for STO

The stereotypes 'Variant' and 'Variation' are used to explicitly specify variability associated with inheritance hierarchies in domain models. The idea is to define a variation point as an abstract class and variants as concrete subclasses [Ziadi and Jezequel, 2006]. Since the subclasses *ClearErrorStatusRequest* and *DiagnosticModeProvideDataReq* in Fig. 3.6 are not mandatory, they are stereotyped 'Variant'. The subclasses *StandardModelProvideDataReq* and *QCModeProvideDateReq* are not stereotyped, thus implying these classes are mandatory for all STO products. We do not use the stereotypes for *Error* and its subclasses since all error types are mandatory in STO. The stereotype 'Optional' is for optional entities which are not part of any inheritance hierarchy (*VoltageDiagnostic* in Fig. 3.6).

Table 3.3 presents some of the use case conditions in Table 3.2 with their corresponding OCL constraints referring to the domain model. Having precise definition of use case conditions is crucial to determine the correct execution of the product in terms of flows of events.

Table 3.3. Some OCL Constraints for STO Use Cases

#	Condition in the Use Case	Corresponding OCL Constraint
1	The operating status is valid	SmartTrunkOpener.allInstances()->forAll (sto|sto.configurationDataStatus = true **and** sto.itsECU.isValid = true **and** sto.itsSTOSensors->forAll (snsr|snsr.isValid = true))
2	Movement is a valid kick	Kick.allInstances()->forAll (k|k.moveAmplitude > k.minKickAmplitude **and** k.moveAmplitude < k.maxKickAmplitude **and** k.moveDuration < 2 **and** (k.backwardMovement - k.forwardMovement) .abs() <= 0.2 **and** k.stopMovement = false **and** (k.timeDisplacementUpperAntenna > k.timeDisplacementLowerAntenna) **and** (k.timeDisplacementUpperAntenna - k.timeDisplacementLowerAntenna) <= 50)
3	There is no error detected	Error.allInstances()->forAll (e|e.isDetected = false)
4	Watchdog reset is valid	SmartTrunkOpener.allInstances()->forAll (sto|sto.resetCounter < 3 **and** sto.itsWatchdog.isEnabled = true)

3.6 Tool Support

We implemented a tool, PUM-C (Product line Use case Modeling - Checker), for checking diagram-specification and specification-template consistency in PUM. PUM-C automatically does the consistency checking and reports inconsistencies such as the diagram missing an include statement in the specification. In addition, it automatically identifies use case conditions (i.e., pre/post conditions and conditions with the keyword 'VALIDATES THAT') and asks the analyst to reformulate them as OCL constraints. To minimize the manual effort, PUM-C first locates conditions in use cases and then identifies repeating and negated ones. If use cases both feature a condition and its negation, the analyst is asked to reformulate only the condition as an OCL constraint. The OCL constraint for the negated condition is automatically derived.

PUM-C relies on NLP and is composed of three layers: *User Interface (UI) Layer*, *Application Layer*, and *Data Layer* (Fig. 3.7). The *UI Layer* supports creating and updating the PUM artifacts. We employ IBM Doors (`www.ibm.com/software/products/ca/en/ratidoor/`) for use case specifications, Papyrus (`https://www.eclipse.org/papyrus/`) for use case diagrams, IBM Rhapsody (`www.ibm.com/software/products/en/ratirhapfami`) for domain models, and Eclipse OCL (`http://www.eclipse.org/modeling/mdt/ocl/`) for writing OCL constraints. To access the *Application Layer* components through the *UI Layer*, we implemented an IBM DOORS plugin.

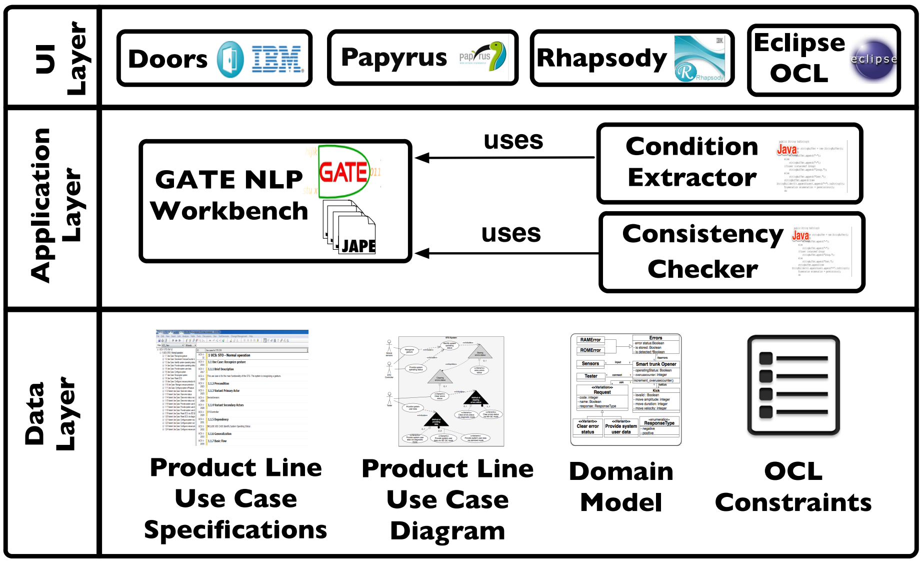


Figure 3.7. Layered Architecture of PUM-C

The *Application Layer* contains the components (i.e., *Condition Extractor* and *Consistency Checker*) which we implemented as Java applications for consistency checking and condition extraction. To perform NLP in use case specifications, these two components use a regular expression engine, called JAPE [H. Cunningham et al, 2018], in the GATE workbench (`http://gate.ac.uk/`), an open-source Natural Language Processing (NLP) framework. JAPE enables to recognise regular expressions in annotations on documents. We implemented the extended RUCM restriction rules in JAPE. In NLP, use cases are first split into tokens. Second, Part-Of-Speech (POS) tags (i.e., *verb*, *noun*, and *pronoun*) are assigned to each token. By using the RUCM restriction rules implemented in JAPE, blocks of tokens are tagged to distinguish RUCM steps (i.e., *output*, *input*, *include*, and *internal operations*) and types of alternative flows (i.e., *specific*, *alternative*, and *global*). The output of the NLP is the annotated use case steps. The *Condition Extractor* and *Consistency Checker* process the annotations and the use case diagram to generate the list of inconsistencies and conditions.

3.7 Industrial Case Study

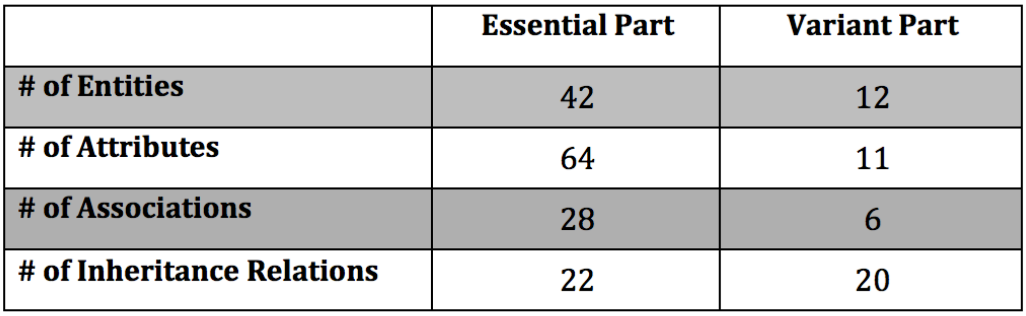
We applied PUM to the functional requirements of STO. Our goal was to assess, in an industrial context, how PUM can improve variability modeling practice and how well PUM addresses the challenges that we identified in capturing requirements variability in use cases. STO was proposed for the assessment by IEE since it was a relatively new project at IEE with multiple potential customers requiring different features. IEE provided their initial STO documentation, which contained a use case diagram, use case specifications, and supplementary requirements specifications describing non-functional requirements. To model the STO requirements according to PUM, we first examined the initial STO documentation and then worked with IEE engineers to build and iteratively refine our models. Tables 3.4 and 3.5 present the size of the resulting use cases and domain model.

Table 3.4. Product Line Use Cases in the Case Study

	# of use cases	# of variation points	# of basic flows	# of alternative flows	# of steps	# of condition steps
Essential Use Cases	15	5	15	70	269	75
Variant Use Cases	14	3	14	132	479	140

To evaluate the output of PUM in light of the challenges we identified earlier, we had a semi-structured interview with five participants holding various roles at IEE (i.e., process manager, software development manager, software lead engineer, system engineer, and software engineer). They all had substantial software development experience, ranging from 8 to 17 years. All participants had experience with use case driven development and modeling. The interview included a presentation illustrating the PUM steps, a tool demo, and detailed examples from STO. The presentation was interactive and included questions posed to the participants about the models for them to take a more active role and give us feedback.

Table 3.5. Size of the Domain Model



	Essential Part	**Variant Part**
# of Entities	42	12
# of Attributes	64	11
# of Associations	28	6
# of Inheritance Relations	22	20

To capture the perception of engineers participating in the interviews, regarding the potential benefits of PUM, and assess how it addresses the targeted challenges, we handed out a questionnaire including questions to be answered according to a Likert scale [Oppenheim, 2005] (i.e., strongly agree, agree, disagree, and strongly disagree). The questionnaire was structured for the participants to assess PUM in terms of adoption effort, expressiveness, comparison with current practice, and tool support. The participants were also encouraged to provide open, written comments.

Results from the interviews showed that all participants agreed on the following positive aspects of PUM:

- The participants considered the extensions to be simple enough to enable communication between analysts and customers. They also stated that the extensions can also be used for internal communication, e.g., for test engineers to perform regression test selection.
- The participants considered the extensions to provide enough expressiveness to conveniently capture variability in their projects. In STO, we were able to capture 8 variation points, 14 variant use cases, and 7 variant dependencies.
- The participants considered the effort required, to learn how to apply PUM and its tool, to be reasonable. They also stated they expect most of the effort to relate to OCL.
- The participants considered PUM to provide better assistance for capturing and analyzing variability information compared to the current, more informal practice in their projects. With PUM, we could unveil variability information not covered in the initial STO documentation. For instance, the use case diagram extensions helped us identify and model that the method of *Clear Error Status via IEE QC Mode* is mandatory while the method of *Clear Error Status via Diagnostic Mode* is optional (see Fig. 3.5), which was not previously documented.
- The participants considered PUM-C to provide useful assistance for minimising inconsistencies in artifacts.

The participants also expressed a number of challenges regarding the application of PUM:

- *Modeling variability in non-functional requirements.* There are numerous types of non-functional requirements (e.g., *security*, *timing*, and *reliability*) which may play a key role in variability associated with functional requirements. It is crucial to capture such aspects as well.
- *Training customers for PUM.* Though the participants considered the effort required to learn PUM to be reasonable, training customers may be more of a challenge. The company may need customers' consent to initiate the modeling effort. Thus, the costs and benefits of PUM should be made clear to customers.
- *Imperfect variability information.* When a new project starts, requirements and their variations might be very difficult to identify. As a result, in the beginning, analysts are expected to redefine

variation points and variants in requirements specifications through frequent iterations. Such changes on variability need to be managed and supported to enable analysts to converge towards consistent and complete requirements and variability information.

- *Adaptations in the tool chain.* PUM-C is currently implemented as a plugin in IBM DOORS in combination with a leading commercial modeling tool used at IEE, i.e., IBM Rhapsody, and Papyrus. PUM-C highly depends on the outputs of these tools. In time, these tools might be replaced with other tools or the newer versions of the same tools. PUM-C needs to be easily adapted for such changes in the tool chain.

Our discussion with participants resulted in the following extensions being required for PUM and PUM-C:

- *Checking consistency between use cases and domain model.* Domain entities identified in a use case may be missing in the domain model. PUM-C can be extended to automatically evaluate the domain model completeness and correctness by checking the mapping between domain entities identified by our NLP application and entities in the domain model (see Chapter 4).
- *Automatic configuration.* A configurator can guide analysts and customers to make decisions regarding variation points, and generate product specific use cases (see Chapter 4).
- *Integrating non-functional requirements with use cases.* Additional extensions can be introduced into PUM to model non-functional requirements, e.g., response time and synchronization requirements.

Threats to validity. The main threat to the validity of our case study regards the generalizability of the conclusions. To mitigate the threat, we applied PUM to an industrial case study that includes nontrivial use cases in an application domain with multiple potential customers and numerous sources of variability. We selected the respondents to our questionnaire and interviews to hold various, representative roles and with substantial industry experience. To limit threats to the internal validity of the case study, we had many interviews with the IEE engineers in the STO project to verify the correctness and completeness of our models.

3.8 Conclusion

This chapter presented a product line methodology centred around use case modeling, called PUM, for documenting variability in use case diagrams and specifications, and associated domain models. Our main motivation was to enable variability modeling by relying exclusively on commonly used artifacts in use-case driven development, thus avoiding unnecessary modeling overhead. We aimed at capturing variability in product lines at a level of granularity enabling both precise communication with various stakeholders, at different levels of details, and guided product configuration. We integrated and adapted existing product line extensions for use case diagrams and domain models, and introduced some template extensions for use case specifications. Initial results from structured interviews with experienced engineers suggested that PUM is accurate and practical to capture variability in industrial settings.

In this chapter, we answered *Research Question 2* : *How to model variability in use case and domain models without additional traceability to feature models? Are there any existing techniques for use case and domain models to model variability? Which techniques can be used?* The product line extensions we employed and proposed in PUM enable the analysts to model variability without requiring feature models and their traces to use cases.

PUM is the first step to achieve our long term objective in this thesis, i.e., automated configuration and change impact analysis in use case driven development and testing within the context of product lines. Chapter 4 presents an automated configuration approach that guides customers in making configuration decisions to automatically generate PS use case and domain models from PL use case and domain models. To address contexts where products are constantly evolving, Chapter 5 presents a change impact analysis approach that helps analysts properly manage changes in configuration decisions.

Chapter 4

Configuration of Product Specific Use Case and Domain Models

In this chapter, we propose, apply, and assess a use case-driven configuration approach which interactively receives configuration decisions from the analysts to automatically generate Product Specific (PS) use case and domain models. Our approach provides the following: (1) a use case-centric product line modeling method (PUM), (2) automated, interactive configuration support based on PUM, and (3) an automatic generation of PS use case and domain models from Product Line (PL) models and configuration decisions. The approach is supported by a tool relying on Natural Language Processing (NLP), and integrated with an industrial requirements management tool, i.e., IBM Doors. We successfully applied and evaluated our approach to an industrial case study in the automotive domain, thus showing evidence that the approach is practical and beneficial to capture variability at the appropriate level of granularity and to configure PS use case and domain models in industrial settings.

4.1 Introduction

Chapter 3 proposed and assessed the Product line Use case modeling Method (PUM), which enables the analysts to capture and document variability in Product Line (PL) use case diagrams, use case specifications, and domain models.

In this chapter, we propose, apply, and assess a use case-driven configuration approach based on PUM. Our goal is to provide a degree of configuration automation that enables effective product-line management in use case-driven development, without requiring additional modeling artifacts and traceability effort. Our approach supports four activities. First, the analysts model the variability information explicitly in a PL use case diagram, its use case specifications, and its corresponding domain model. Second, the consistency of the PL use case diagram and specifications are checked and inconsistencies are reported if there are any. For instance, a variation point in the use case diagram might be missing in the corresponding use case specification or a use case specification may not conform to the extended RUCM template. Third, the analyst is guided to make configuration decisions based on variability information in the PL models. The partial order of decisions to be made is automatically identified from the dependencies among variation points and variant use cases. In the case of contradicting configuration decisions, such as two decisions resulting in selecting variant use cases violating some dependency constraints, we automatically detect and report them. The analyst must then backtrack and revise the decisions to resolve these inconsistencies. Alternatively, we could employ constraint solvers (i.e., SAT solver, BDD solver and Prolog solver) to identify a priori possible contradicting decisions so as to avoid them. However, according to our observation at IEE, customers are also involved in the decision-making process in which they frequently re-evaluate, backtrack and revise their decisions. Therefore, it is important for them to have the possibility to make contradicting decisions and revise prior ones as a result. Fourth, based on configuration decisions, the Product Specific (PS) use case and domain models are automatically generated from the PL use case and domain models. To support these activities, we developed a tool, *PUMConf (Product line Use case Model Configurator)*. The tool automatically checks the consistency of the PL models, identifies the partial order of decisions to be made, determines contradicting decisions, and generates PS use case and domain models. To summarize, the contributions of this chapter are:

- a configuration approach that is specifically tailored to use case-driven development, and that guides the analysts and customers in making configuration decisions in product lines to automatically generate PS use case and domain models;

- tool support integrated with an industrial requirements management tool (i.e., IBM Doors) as a plug-in, which relies on Natural Language Processing (NLP) to report inconsistencies in PL use case models and contradicting configuration decisions, and to automatically generate PS use case and domain models;

- an industrial case study demonstrating the applicability and benefits of our configuration approach.

In this chapter, we answer *Research Question 3* raised in Chapter 1: *To what extent and how can we automate the interactive configuration of use case and domain models? How can we support the analysts for making configuration decisions and for generating PS use case and domain models?* With the configuration approach and its contradiction identification feature, we address the issues about interactive decision making and generation of PS use case and domain models.

This chapter is structured as follows. Section 4.2 introduces the industrial context of our case study to illustrate the practical motivations for our configuration approach. Section 4.3 discusses the related work in light of our needs. In Section 4.4, we provide an overview of the approach. In Section 4.5, we illustrate our approach through example models. In Sections 4.6 and 4.7, we provide the details of the core technical parts of our approach: consistency checking of configuration decisions and generation of PS use case and domain models. Section 4.8 presents our (publicly available) tool support for configuration, while Section 4.9 presents our industrial case study, i.e., Smart Trunk Opener (STO), along with results. We conclude the chapter in Section 4.10.

4.2 Motivation and Context

Our configuration approach is developed in the context of embedded software systems interacting with multiple external systems, configured for multiple customers, and developed according a use case-driven process. In such a context, requirements variability is communicated to customers and an interactive configuration process is followed for which guidance and automated support are needed. For instance, for each product in a product family, IEE negotiates with customers how to resolve variation points in requirements, in other words how to configure the product line.

Within our context, we identify two challenges that should also apply to other environments and that need to be considered in reusing use cases and a domain model for a product family:

Challenge 1: **Modeling Variability Information with Least Possible Modeling Overhead.** The analysts need to explicitly document variability information (e.g., variant use cases, variation points, and optional steps) for use cases and a domain model to be communicated to the customers during product configuration. Relating feature models to use cases and domain model is the most straightforward option but has shortcomings in terms of additional modeling and traceability effort. For example, it would not be easy for analysts and customers to comprehend and visualize all variability information traced to use case diagram, use case specifications and domain model. In STO, we identified 15 mandatory and 14 variant use cases which contain 8 variation points and 7 variant dependencies. The STO use cases include 244 use case flows (29 basic flows, 202 alternative flows, and 13 optional alternative flows) with 27 optional steps while the STO domain model contains 11 variant domain entities. The variability information scattered across all these use case flows with trace links from feature models would be communicated to customers and used to configure a product.

Challenge 2: **High Degree of Automation in Use Case-Driven Configuration.** In order to facilitate use case-driven configuration in industrial practice, a high degree of automation is a must while the analysts are interactively guided for their decisions. Before the configuration, it should be automatically confirmed that all artifacts with variability information, including use case diagram, specifications, and domain model, are consistent. Any inconsistency in these artifacts may cause invalid configuration outputs. Adding to the complexity affecting the decision-making process during configuration, there may be contradicting decisions and hierarchies among decisions. During the configuration process, the analysts need to be interactively informed about contradicting decisions and about the order of possible decisions. With interactive guidance and proper tool support, the analysts can fix inconsistent PL artifacts and resolve decision contradictions, which leads to the automatic generation of PS use cases and their domain model.

We already addressed the first challenge in Chapter 3 with the *Product line Use case modeling Method (PUM)*. In the remainder of this chapter, we focus on how to best address the second challenge in a practical manner, in the context of use case-driven development, while relying on PUM to minimize the modeling overhead.

4.3 Related Work

In this section, we cover the related work across three categories.

Configuration Techniques for Requirements Variability. Eriksson et al. [Eriksson et al., 2009] provide an approach to manage natural-language requirements specifications in the software product line context. Variability is captured and managed using a feature model while requirements in various forms, e.g., use cases, textual requirements specifications and domain model, are traced to the feature model. The analyst selects the features in the feature model to be included in the product. By following traces from the selected features to the requirements, the approach filters those requirements that are relevant for a specific product in the product line. These filtered requirements are then exported as product requirements specifications. The approach does not support any automated decision-making solution (e.g., decision ordering, decision consistency checking, and inferring decisions) for selecting features (*Challenge 2*). In addition, the analyst has to manually assign traces between features and requirements at a very low level of granularity, i.e., sequences of use case steps (*Challenge 1*). pure::variants [Pur, 2018b] is a tool to manage all parts of software products with their components, restrictions and terms of usage. Its extension, pure::variants for IBM DOORS [Pur, 2018a], enables the analyst to capture variability as features in a feature model, and trace them to requirements specifications in IBM DOORS. It transforms the requirements specifications into product requirements based on the selected features in the feature model. Compared to the approach proposed by Eriksson et al. [Eriksson et al., 2009], pure::variants for IBM DOORS provides a better automated support, i.e., an automated contradiction detection for feature models. However, the analyst still suffers from the same modeling overhead, when pure::variants is employed. The analyst needs to manually establish traces at a very low level granularity and maintain these traces when the feature model or requirements specifications evolve. There are similar approaches, eg., [Gomaa, 2000, seok Choi et al., 2008, Buhne et al., 2006, Alferez et al., 2008, Braganca and Machado, 2007], which require modeling and maintenance overhead with poor automated configuration support. Our approach attempts to minimize this overhead by capturing variability information in use case and domain models (*Challenge 1*).

Moon et al. [Moon et al., 2005, Moon and Yeom, 2004] propose a method that generates PS use cases from PL use cases without using any feature model. However, the proposed method requires that Primitive Requirements (PR) (i.e., building blocks of complex requirements) be specified by the analyst and traced to the use case diagram and specifications via the *PR - Context* and *PR - Use Case* matrices. The analyst has to manually encode traceability information in these matrices (*Challenge 1*). The configuration takes place by selecting the *PR*s in the matrices without any automated decision-making support (*Challenge 2*).

John and Muthig [John and Muthig, 2004] introduce some product line extensions to use case diagrams and specifications to be able to capture variant use cases without a feature model. They propose a new artifact, called decision model, to represent variation points textually in a tabular form. Each variation point has multiple facts which represent decisions. For each decision, there are actions which describe configuration operations for use cases, e.g., removing parts of a use case. The analyst is expected to configure, with the help of the decision model, the product specific use case diagram and specifications but such a decision model can quickly become too complex for the analyst to comprehend. There is no automated tool support reported for the approach (*Challenge 2*). Faulk [Faulk, 2001] proposes the use of a similar decision model to generate PS requirements specifications from PL requirements specifications. Biddle et al. [Biddle et al., 2002] provide support for configuring use case specifications through parametrization. Parameters can be specified anywhere in the name or body of a parameterized use case. The manual assignment of values to parameters is considered as configuring product specific use case specifications (*Challenge 2*). Fantechi et al. [Fantechi et al., 2004b, Fantechi et al., 2004a] propose Product Line Use Cases (PLUC), an extension of the Cockburn use case template with three kinds of tags (i.e., alternative, parametric, and optional). It is not possible with these tags to explicitly represent mandatory and optional variants. Variants and variation points are hidden in use case specifications conforming to PLUC. These two approaches [Biddle et al., 2002, Fantechi et al., 2004b] do not support variability in the use case diagram and domain model. They also lack automated support for the decision-making process including decision ordering and detection of contradicting decisions (*Challenge 2*).

Annotation- and Composition-based Configuration for Scenario-based Requirements. Some approaches specialize in configuring scenario-based requirements using annotation- and composition-based techniques [Alférez et al., 2014]. The Product Line Use case modeling for Systems and Software engineering approach (PLUSS) proposed by Eriksson et al. [Eriksson et al., 2009, Eriksson et al., 2004, Eriksson et al., 2005a] uses feature models to configure requirements in multiple forms including scenario-based requirements models (e.g., use cases and activity diagrams). PLUSS employs annotations throughout requirements to represent how they are related to features. Czarnecki and Antkiewicz [Czarnecki and Antkiewicz, 2005] propose another configuration approach based on annotation of scenarios using feature models. Activity diagrams are used to specify scenarios. Traces between feature models and activity diagrams are given as special annotations on activity diagrams. To annotate activity diagrams, the approach employs model templates, which contain the union of the model elements, e.g., presence conditions and meta-expressions, in all valid template instances, i.e., annotated activity diagrams. A product is specified by creating a feature configuration based on the feature model. The model template is instantiated automatically by using the feature configuration. The generated template instance is an activity diagram of the specified product. Although the template instantiation is automated, feature configuration is manual (*Challenge 2*). The analyst also has to manually create a feature model and a model template for annotations.

Bonifácio et al. [Bonifácio et al., 2008, Bonifácio and Borba, 2009] propose a framework for modeling the composition process of scenario variability mechanisms (MSVCM). They provide a *weaver* (configurator) that takes a PL use case model, a feature model, a product configuration, and configuration knowledge as input. The product configuration artifact identifies a specific product, which is characterized by a configuration of features in the feature model, while the configuration knowledge relates features to transformations used for automatically generating the PS use case model. These two artifacts are manually created by the analyst (*Challenge 2*). The Variability Modeling Language for Requirements (VML4RE) [Alférez et al., 2009, Zschaler et al., 2009] presents a similar solution for the composition of use case diagrams and their selected scenarios represented by activity diagrams. It supports the definition of traces between feature models and requirements (e.g., use case diagram and activity diagram). VML4RE provides a simple set of operators to specify the composition of requirements models for generating PS requirements models. There are more composition-based approaches [Varela et al., 2011, Mussbacher et al., 2012, Blanes et al., 2014] to configure scenario-based requirements using feature models.

All these configuration approaches given above require additional modeling and traceability effort for feature models (*Challenge 1*) while most of them do not provide a high degree of automation for the decision-making process (*Challenge 2*). There are approaches [Stoiber et al., 2010, Wang et al., 2009, Weston et al., 2009] that support the identification and extraction of variable features from given requirements but these approaches still require a considerable manual intervention in the identification of features. In addition, the detailed functionality of a feature is still shown in the traced requirements documents, and this requires frequent context switching, which is not practical in industrial projects. Stoiber and Glinz [Stoiber and Glinz, 2010] propose the modularization of variability information in decision tables to avoid context switching, but the analyst still needs to manually encode all decision constraints and traces in such tables, which can easily get too complex to comprehend (*Challenge 1*). Bonifácio et al. [Bonifácio et al., 2015] argue that annotation-based approaches entangle the representation of common and variant behavior, whereas the composition-based approaches provide a better separation of variant behavior. They compared an annotation-based approach, i.e., PLUSS, with a composition-based approach, i.e., MSVCM, to investigate whether the composition-based approach causes extra costs for modularizing scenario specifications. They concluded that although MSVCM improves modularity, it requires more time to derive PL specifications, and more investments on training.

Configuration Tools. Nie et al. [Nie et al., 2013] describe the key automation functionalities that configuration tools should support: *inferring decisions*, *consistency checking*, *decision ordering*, *collaborative configuration*, and *reverting decisions* (*Challenge 2*). Our tool, *PUMConf*, automatically infers new configuration decisions based on prior decisions and variation point-variant dependencies. The consistency of all inferred and prior decisions are automatically checked. The analyst can also revert configuration decisions in order to maintain the consistency. *PUMConf* provides decision ordering to guide the analyst in which sequence a set of decisions should be made, by taking into account the hierarchies among variation points. Currently, *PUMConf* does not support collaborative configuration in terms of PS use case models. Collaborative configuration is defined as coordinating the configuration of multiple systems where the configuration of one system depends on the configuration of other systems [Nie et al., 2013]. We need to extend our PL use case modeling method in such a way that the analyst is able to model dependencies among PL use case models of multiple systems. Our tool can then be extended to support collaborative configuration using such dependencies.

Configuration tools in the literature partially support the key automation functionalities within a context not specific to use case-driven configuration (*Challenge 2*). Le Rosa et al. [Rosa et al., 2009] provide a questionnaire-based system configuration tool to capture system variability based on questionnaire models composed of questions that refer to a set of *facts* to be set to true or false. When the questionnaire is answered by the analyst, the tool assigns values to facts, and derives an individualized system by using the resulting valuation. The tool supports the key functionalities, except collaborative configuration. Another configuration tool is C2O, presented by Nohrer et al. [Nöhrer and Egyed, 2013, Nöhrer et al., 2012, Nöhrer and Egyed, 2010]. The tool enables the analysts to make configuration decisions in an arbitrary order while it guides them by rearranging the order of decisions (decision ordering), inferring decisions to avoid follow-on conflicts (inferring decisions), and provides support in fixing conflicts at a later time (consistency checking and reverting decisions). No support for collaborative configuration is reported for C2O. Myllarniemi et al. [Myllärniemi et al., 2005] present Kumbang, a prototype configurator for product individuals from configurable software product families. The tool focuses on the configuration of architecture models. It supports the key functionalities except inferring decisions and collaborative configuration. SPLOT [Mendonca et al., 2009a] is a web-based configurator benefiting from SAT solvers and binary decision diagrams to support reasoning and interactive configuration on feature models. COVAMOF [Sinnema and Deelstra, 2008, Sinnema et al., 2004] supports only architecture configuration, while DOPLER [Dop, 2018, Dhungana et al., 2011] is a more general configurator which can be customized for multiple artifacts such as components, test cases, or documentation fragments.

The tools given above are either general configurators (e.g., [Dop, 2018, Dhungana et al., 2011, Rosa et al., 2009]) or custom configurators for artifacts such as architecture and feature models (e.g., [Myllärniemi et al., 2005, Sinnema and Deelstra, 2008, Sinnema et al., 2004]), which are quite different than our target artifacts. General configurators could be employed to configure PS use case and domain models. For instance, DOPLER [Dop, 2018, Dhungana et al., 2011] supports capturing the variability information using decision models and modeling any type of artifact as asset models. Decision and assets are linked by using traceability relations. The analyst has to model variability information in a decision model by using DOPLERVML, a modeling language for defining product lines. Even if use cases and domain models can automatically be translated into an asset model, the analyst still has to manually encode the decisions in the decision model and assign the traceability relations between decision and asset models, which are some inclusion links. Having all these decision and asset models with their explicit traces is exactly the type of modeling practice that we try to avoid in our configuration approach (*Challenge 1*). In addition, DOPLER requires considerable effort and tool-specific internal knowledge to be customized for the consistency checking of configuration decisions and generation of PS use case and domain models.

4.4 Overview of Our Approach

The process in Fig. 4.1 presents an overview of our configuration approach. In Step 1, *Elicit product line use case and domain models*, the analyst elicits PL use cases and a domain model with the use case diagram, the RUCM template, and their product line extensions.

Step 1 is manual. Its output includes (1) a *PL use case diagram*, which captures variability, and its constraints and dependencies, (2) *PL use case specifications*, which detail the variability information captured in the diagram, and (3) a *PL domain model*, which captures variability in domain entities (*Challenge 1*). In Step 2, *Check consistency of product line use case and domain models*, our approach automatically checks the consistency of use case diagram, use case specifications (also with the RUCM template), and domain model to report inconsistencies (*Challenge 2*). Steps 1 and 2 are iterative: the PL diagram, specifications, and domain model are updated until full consistency is achieved. We already discussed these two steps in Chapter 3.

In Step 3, *Configure product specific use case and domain models*, the user is asked to input configuration decisions regarding variation points captured in PL use case and domain models to automatically configure the product line into a product. The configuration step is the main focus of this chapter. It is described in Section 4.5.

Step 3 includes an automated, iterative, and interactive decision-making activity (*Challenge 2*). The partial order of configuration decisions to be made is automatically identified from the dependencies among variation points and variant use cases. The analyst is asked to input the configuration decisions in the given partial order. When a decision is made, the consistency of the decision with prior decisions is checked. There might be contradicting decisions in the PL use case diagram such as two decisions resulting in selecting variant use cases violating some dependency constraints. These are automatically determined and reported *a posteriori* and the analyst can backtrack and revise his

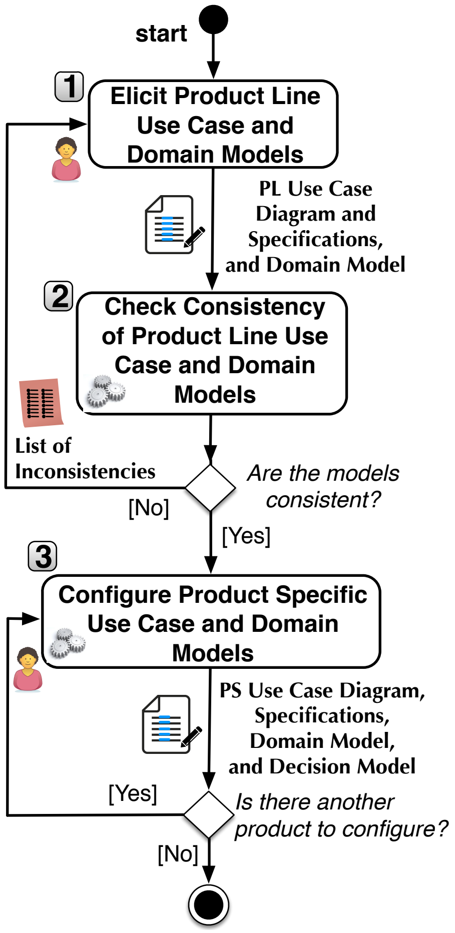

start
1
Elicit Product Line Use Case and Domain Models
PL Use Case Diagram and Specifications, and Domain Model
2
Check Consistency of Product Line Use Case and Domain Models
List of Inconsistencies
Are the models consistent?
[No]
[Yes]
3
Configure Product Specific Use Case and Domain Models
PS Use Case Diagram, Specifications, Domain Model, and Decision Model
[Yes]
Is there another product to configure?
[No]

Figure 4.1. Overview of the Approach

decisions. Alternatively, the analyst could be guided through the configuration space in such a way that decisions that may lead to contradictions are avoided *a priori* [Rosa et al., 2009]. SAT solvers can be employed to incrementally prune the configuration space in an interactive configuration [Batory, 2005] while CSP solvers are used to handle additional modeling elements in terms of variables (e.g., sets and finite integer domains) and constraints (not only propositional formulas) [Benavides et al., 2005b, Benavides et al., 2010]. However, the use of the SAT and CSP solvers in an iterative and interactive product configuration can be challenging since (a) it can quickly become infeasible to compute inferences, which dynamically prune the configuration space, when the number of variables to be computed is large and (b) it may require considerable implementation effort and internal tool knowledge to use these solvers for computing inferences and detecting and reporting contradictions [Batory, 2005, Forbus and Kleer, 1993]. According to our observation in industry, customers are also involved in the decision-making process. They need to account for the entire configuration space, including contradicting decisions, because they frequently re-evaluate decisions and possibly update them. Therefore, we decided to have an *a posteriori* approach for the consistency checking of configuration decisions, and implemented our own algorithm fitting our context.

Our motivation is to rely, to the largest extent possible, on a solution that can be easily customized for further extensions addressing automated reconfiguration, including change impact analysis for evolving configuration decisions on PL use case and domain models (see Chapter 5). One may argue that existing configuration tools [Rosa et al., 2009, Sinnema et al., 2004, Nöhrer and Egyed, 2013, Czarnecki et al., 2005] could be reused within the context of use case-driven configuration but these tools either need feature models [Czarnecki et al., 2005] or require the variability information to be depicted independently of specific notations or languages, by means of a set of facts [Rosa et al., 2009, Nöhrer and Egyed, 2013]. As discussed earlier, not only does this not match our practical needs but, furthermore, we also need a custom solution to generate PS use case and domain models based on the decisions. The entire configuration approach is illustrated by an example in Section 4.5. We provide the details of our decision consistency checking algorithm in Section 4.6, whereas Section 4.7 presents the generation of PS use case and domain models from PL models.

4.5 Configuration of Product Specific Use Case and Domain Models

The product configuration is a decision-making process, where the variability information is examined to select the desired features for the product. A product in a product family is defined as a unique combination of features selected during configuration. In this chapter, we rely on variability information given in the PL use case diagram, specifications and domain model. The user selects (1) the desired use cases in the PL use case diagram, (2) the use case elements in the PL use case specifications, and (3) the domain entities in the PL domain model, to generate the PS use case diagram, specifications, and domain model.

Our configuration mechanism relies on a use case configuration function. The configuration function takes a PL use case diagram, a set of PL use case specifications, and a PL domain model as input, and produces a PS use case diagram, a set of PS use case specifications, a PS domain model, and a decision model which captures configuration decisions. Such a decision model is important since the analyst/customer may need to update decisions to reconfigure the PS models for the same product.

The decision model conforms to a decision metamodel, which is described in Fig. 4.2. We will shortly describe its elements.

There are four main use case elements for which the user has to make decisions (i.e., *Variation Point*, *Optional Step*, *Optional Alternative Flow*, and *Variant Order*). In a variation point, the user selects variant use cases to be included for the product. For PL use case specifications, the user selects optional steps and alternative flows to be included and determines the order of steps (variant order). In the PL domain model, the user makes decisions for the *Variant* and *Optional* entities. All these decisions are saved in the decision model, whose structure is formalized by the decision metamodel.

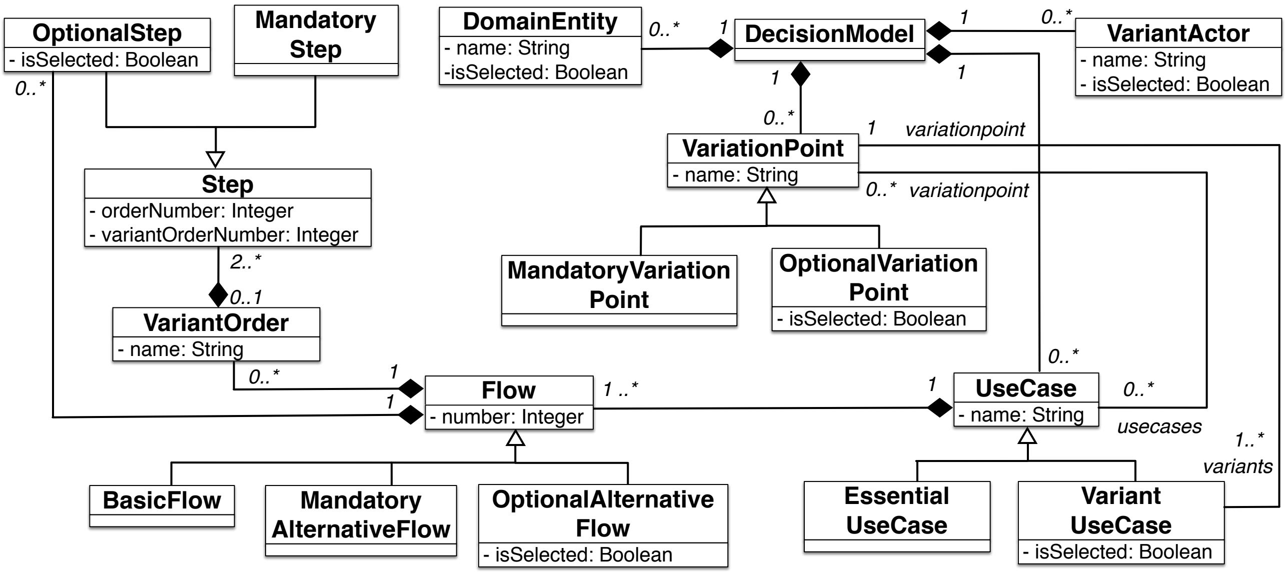


Figure 4.2. Decision Metamodel

The reader is referred to Appendix A for the algorithm of the configuration function. At a high level, the algorithm first traverses the use case diagram for variation points, and then processes use case specifications for optional alternative flows, optional steps, and variant orders. Then the domain model is traversed for *Variant* and *Optional* domain entities. In the following, we explain the steps of the algorithm with an illustrative example. The example is a slight adaptation of part of our industrial case study since we needed some additional modeling elements to illustrate the complete set of features of the algorithm. Fig. 4.3 depicts an example PL use case diagram with four variation points, eight variant use cases, and one essential use case.

The main steps of the configuration algorithm for use case diagrams are:

- *Identifying variation points in the diagram to start the configuration.* In Fig. 4.3, there are four variation points and two of them are included by variant use cases in another variation point (i.e., *UC2* in *VP1* includes *VP2*, and *UC3* includes *VP3*). The algorithm automatically filters out the variation points included by variant use cases because the analyst can make a decision for these variation points only if the including variant use case is selected for the product. For instance, *VP2* will not be considered if *UC2* is not selected in *VP1*. The user can start making decisions with either *VP1* or *VP4*.

- *Getting a decision for each variation point and resolving contradicting decisions.* This is an iterative step. For each variation point identified, the analyst is asked to make a decision. A decision is about selecting, for the product, variant use cases in the variation point. After the analyst makes the decision, the algorithm first checks the associated cardinality constraints. The use case diagram is then traversed to determine previous decisions contradicting the current decision. If there is any contradiction, the analyst is expected to update one or more decisions to resolve the contradiction.

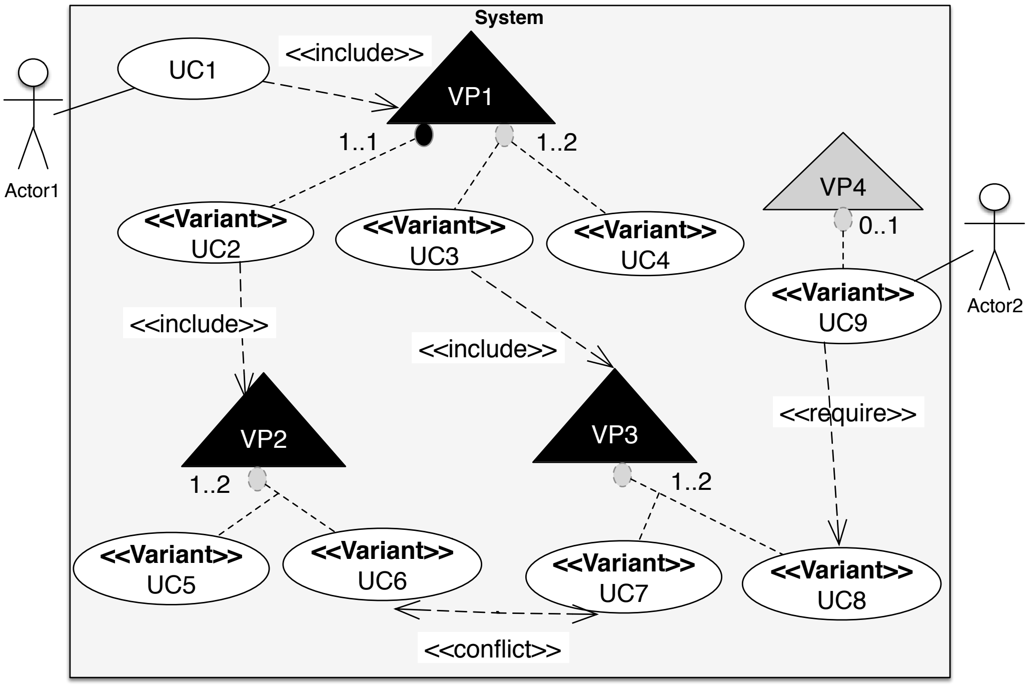


Figure 4.3. Example Product Line Use Case Diagram

- The analyst makes a decision in *VP4*, which is selecting *UC9* for the product. The decision complies with the cardinality constraint. No contradiction is identified since there is no previous decision. Therefore, the decision is saved in the decision model.
- The analyst proceeds with *VP1*. *UC2* and *UC4* are selected while *UC3* is not selected for the product. The decision complies with the cardinality constraint in *VP1*. The algorithm identifies a contradiction with the decision in *VP4*. Since the user does not select *UC3*, *UC7* and *UC8* in *VP3* included by *UC3* are also automatically not selected. However, *UC9* requires *UC8* in the product. The analyst is asked to resolve the contradiction by updating either the decision for *VP4* or the decision for *VP1*.
- The analyst updates the decision in *VP1* to resolve the contradiction. Only *UC2* and *UC3* are selected in the updated decision, which complies with the cardinality constraint. At this point, there is no contradiction identified. *UC3* is already selected but the analyst has not yet made the decision for *VP3*. Therefore, the decision is saved in the decision model.
- The analyst is asked to make further decisions for the variation points included by the selected variant use cases. *UC2* and *UC3* include *VP2* and VP3, respectively. In the decisions for *VP2* and *VP3*, the analyst selects *UC6* and *UC8*. The decisions comply with the associated cardinality constraints, and there is no contradiction identified with the previous decisions. While *UC6* is selected, the user decides not to have *UC7* (UC6 conflicts with UC7 in Fig. 4.3). *UC9* and *UC8* are selected in *VP4* and *VP3*, respectively (UC9 requires UC8). The decisions are saved in the decision model. All variation points are addressed after the analyst selects *UC2*, *UC3*, *UC6*, *UC8*, and *UC9*.

- *Generating the PS use case diagram from the PL diagram.* By using the decisions stored in the decision model, the algorithm automatically generates the PS use case diagram from the PL diagram (see Fig. 4.4 for the PS diagram generated from Fig. 4.3). The transformation is based on a set of transformation rules further described in Section 4.7. For instance, for *UC1*, *VP1* and the selected *UC2* in Fig. 4.3, the algorithm creates *UC1* and *UC2* with an *include* relation in Fig. 4.4.

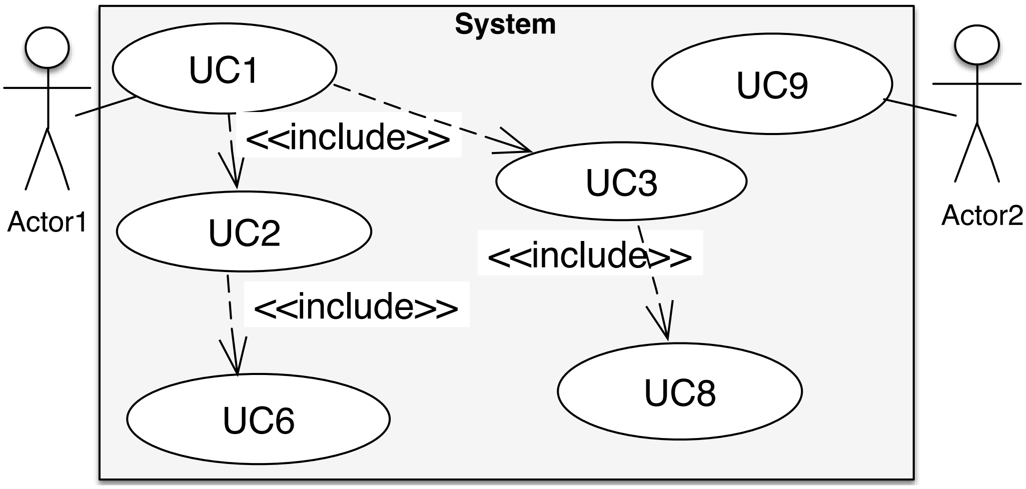


Figure 4.4. Generated Product Specific Use Case Diagram

One may argue that the *include* relations with the single, including use cases in Fig. 4.4 are redundant because an *include* relation is used to show that the behavior of an included use case is inserted into the behavior of multiple including use cases. Alternatively, we could choose an approach which directly inserts (copy-and-paste) the behavior of the included use cases into the including use case specifications. However, to ease the traceability between PL and PS use case diagrams, we prefer to employ include relations in the PS use case diagram even if they are for single, including use cases. For instance, *UC2*, *UC3*, *UC6* and *UC8* in Fig. 4.4 can be directly traced to the variant use cases *UC2*, *UC3*, *UC6* and *UC8* in Fig. 4.3, respectively.

After the PL use case diagram, the algorithm handles the PL use case specifications. Table 4.1 provides the PL specifications of some use cases in Fig. 4.3. The algorithm has two steps for use case specifications:

- *Getting decisions for each optional step, optional alternative flow, and variant order group.* In Table 4.1, there are two variation points (Lines 5 and 36), one variant use case (Lines 21-37), two optional steps (Lines 24 and 32), one optional alternative flow (Lines 8-12), and one variant order group (Lines 23-25). The decisions for variant use cases have already been made in the PL diagram (selecting *UC2* and *UC3* in *VP1* and *UC6* in *VP2*). Therefore, in this step, the analyst is only asked to make decisions for optional steps, optional alternative flows, and variant order groups. For example, the user selects only one of the optional steps (Line 24) with

Table 4.1. Some Example Specifications of the PL Use Cases in Fig. 4.3

1	**USE CASE** UC1
2	**1.1 Basic Flow**
3	1. The system VALIDATES THAT the operating status is valid.
4	2. The system REQUESTS the measurements FROM the sensors.
5	3. INCLUDE <VARIATION POINT: VP1>.
6	4. The system VALIDATES THAT the computed value is valid.
7	5. The system SENDS the computed value TO the STO Controller.
8	**1.2 <OPTIONAL>Bounded Alternative Flow**
9	RFS 1-4
10	1. IF voltage fluctuation is detected THEN
11	2. RESUME STEP 1.
12	3. ENDIF
13	**1.3 Specific Alternative Flow**
14	RFS 1
15	1. ABORT.
16	**1.4 Specific Alternative Flow**
17	RFS 4
18	1. The system increments the counter by the increment step.
19	2. ABORT.
20	
21	**<VARIANT>USE CASE** UC2
22	**1.1 Basic Flow**
23	V1. The system SENDS measurement errors TO the STO Controller.
24	V2. <OPTIONAL>The system VALIDATES THAT RAM is valid.
25	V3. The system VALIDATES THAT the sensors are valid.
26	4. The system VALIDATES THAT there is no error detected.
27	**1.2 Specific Alternative Flow**
28	RFS V2
29	1. ABORT.
30	**1.3 Specific Alternative Flow**
31	RFS V3
32	1. <OPTIONAL>The system SENDS diagnosis TO the STO Controller.
33	2. ABORT.
34	**1.4 Specific Alternative Flow**
35	RFS 4
36	1. INCLUDE <VARIATION POINT: VP2>.
37	2. ABORT.

the order *V2*, *V1*, and *V3* (Lines 23-25). The optional bounded alternative flow is not selected. These decisions are saved in the decision model.

- *Generating PS use case specifications from PL use case specifications.* The PL use case specifications are automatically transformed into the PS specifications based on configuration decisions and a set of transformation rules (see Section 4.7). Table 4.2 provides some of the PS use case specifications generated from Table 4.1. First, some of the variation points in the PL specifications (Lines 5 and 36 in Table 4.1) are transformed based on the decisions for the PL

Table 4.2. Some of the Generated PS Use Case Specifications

1	**USE CASE** UC1
2	**1.1 Basic Flow**
3	1. The system VALIDATES THAT the operating status is valid.
4	2. The system REQUESTS the measurements FROM the sensors.
5	3. The system VALIDATES THAT ‘Precondition of UC2’.
6	4. INCLUDE UC2.
7	5. The system VALIDATES THAT the computed value is valid.
8	6. The system SENDS the computed value TO the STO Controller.
9	**1.2 Specific Alternative Flow**
10	RFS 1
11	1. ABORT.
12	**1.3 Specific Alternative Flow**
13	RFS 3
14	1. INCLUDE UC3.
15	2. RESUME STEP 5.
16	**1.4 Specific Alternative Flow**
17	RFS 5
18	1. The system increments the counter by the increment step.
19	2. ABORT.
20	
21	**USE CASE** UC2
22	**1.1 Basic Flow**
23	1. The system VALIDATES THAT RAM is valid.
24	2. The system SENDS measurement errors TO the STO Controller.
25	3. The system VALIDATES THAT the sensors are valid.
26	4. The system VALIDATES THAT there is no error detected.
27	**1.2 Specific Alternative Flow**
28	RFS 1
29	2. ABORT.
30	**1.3 Specific Alternative Flow**
31	RFS 3
32	1. ABORT.
33	**1.4 Specific Alternative Flow**
34	RFS 4
35	1. INCLUDE UC6.
36	2. ABORT.

diagram. For instance, for *VP1*, the configurator creates two include statements for *UC2* and *UC3* (Lines 6 and 14 in Table 4.2) with a validation step (Line 5 in Table 4.2) and a corresponding specific alternative flow where *UC3* is included (Lines 12-15). The created validation step checks if the precondition of *UC2* is met. If the condition holds, *UC2* is executed in the basic flow (Line 6). If not, the newly created alternative flow is taken and *UC3* is executed (Line 14). For *VP2*, only *UC6* is included in the PS specification of *UC2* since the user selects only *UC6*. After handling variation points, selected optional steps and optional alternative flows are included in the PS specifications (Line 23). Variant order groups are ordered in the

PS specifications according to the given decision (Lines 23-26).

Lastly, the configuration algorithm collects decisions for each optional and variant domain entities in order to generate the PS domain model from the PL domain model. The algorithm does not currently take into account the variant dependencies in PL domain models. It would need further extensions for detecting configuration decisions violating constraints imposed by variant dependencies. The detection of such decisions in PL domain models is very similar to the consistency checking of configuration decisions in PL use case diagrams (see Section 4.6.3). Therefore, we would only need to adapt the current consistency checking algorithm for PL domain models. The PS domain model generation is straightforward. In addition to mandatory entities, each selected optional and variant entity is copied into the PS domain model without product line stereotype. The generated model is pruned for the cases where a *is-a* relation has only a single subclass. Fig. 4.5 provides the PS domain model generated from the PL domain model in Fig. 3.6.

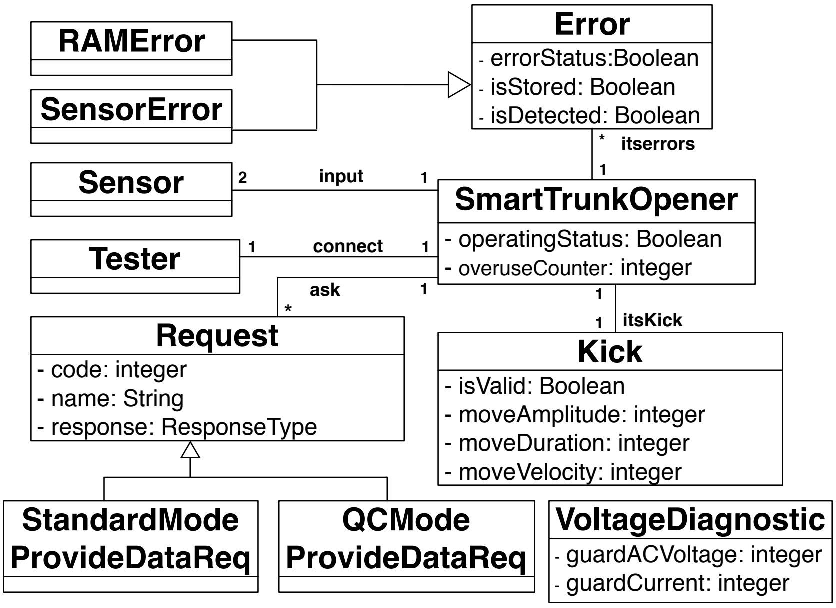


Figure 4.5. Generated Product Specific Domain Model

The PL domain model contains two variant entities (*ClearErrorStatusRequest* and *DiagnosticModeProvideDataReq*) and one optional entity (*VoltageDiagnostic*). The analyst selects only *VoltageDiagnostic* for the PS domain model. Therefore, only *VoltageDiagnostic* among variant and optional entities is copied into the PS domain model in Fig. 4.5. In the PL model, there are two main request types, i.e., *ClearErrorStatusRequest* and *ProvideSystemDataRequest*, with sub request types. Since *ClearErrorStatusRequest* is not selected by the analyst, *ProvideSystemDataRequest* is the only remaining subclass of *Request* in the inheritance hierarchy. During pruning of the PS model, it is removed and the subclasses *StandardModeProvideDataReq* and *QCModeProvideDataReq* become direct specializations of *Request*.

Another output of our configuration approach is a decision model which conforms to the decision metamodel in Fig. 4.2. The decision model stores all the configuration decisions for the use case diagram, the use case specifications, and the domain model. Fig. 4.6 depicts the decision model resulting from the example configuration using the PL use case diagram in Fig. 4.3, the PL use case specifications in Table 4.1, and the PL domain model in Fig. 3.6.

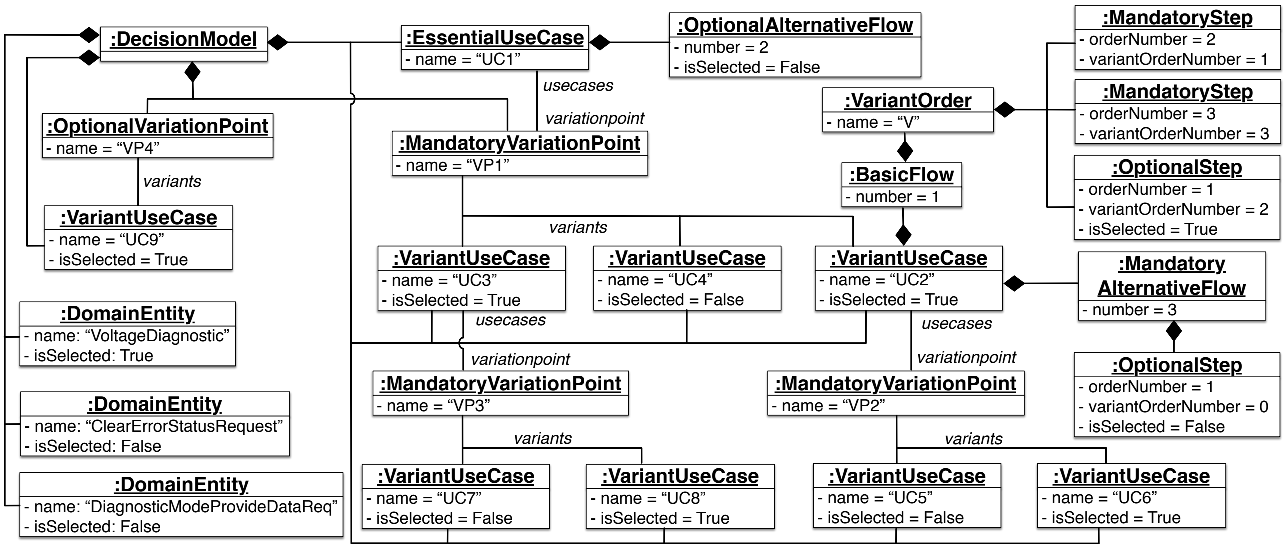


Figure 4.6. Example Decision Model Resulting from the Example Configuration

The decision model in Fig. 4.6 contains several instances of *DomainEntity*, *VariationPoint*, *UseCase* for which the analyst made the decisions described above. For instance, the *VP1* instance of *MandatoryVariationPoint* is associated with three instances *UC2*, *UC3* and *UC4* of *VariantUseCase*, while the decision for the variation point is encoded as *True* or *False* using the *isSelected* attribute. The decision models can be employed for further use such as representing decisions for reconfiguration of the same product and comparison of the decisions in multiple products for regression test selection within the context of product line testing, as further described in Chapters 5 and 6.

4.6 Consistency Checking of Configuration Decisions

In this section, we present the details of the consistency checking of configuration decisions in the PL use case diagram. The objective of consistency checking (Section 4.6.1) is to identify contradicting decisions for variation points in a configurable PL use case diagram (Section 4.6.4). The consistency checking algorithm (Section 4.6.3) is based on mapping from the PL use case diagram to propositional logic (Section 4.6.2).

4.6.1 Objective and Assumptions

Consistency checking of configuration decisions is vital at collecting decisions from the analyst. PS models cannot be generated from inconsistent decisions. In our configuration approach, consistency checking aims at determining contradicting decisions for the variation points in the PL use case diagram. Two or more configuration decisions may contradict each other if they result in violating some variation point and variant dependency constraints (i.e., *require* and *conflict*). Assume there are two conflicting, variant use cases *Ua* and *Ub* (i.e., *Ua* conflicts with *Ub*). *Ua* and *Ub* are selected in decisions *Da* and *Db*, respectively. *Da* and *Db* are contradicting because *Ua* and *Ub* cannot exist for the same product.

The analyst needs to update decisions in order to resolve contradictions. Our approach follows the *Fix right away with selective (multiple) undo* strategy [Nöhrer and Egyed, 2010] in which only involved decisions are updated to return the configuration to a consistent state immediately when the analyst introduces a contradicting decision. To do that, we automatically identify the decisions involved in the contradiction. We chose this strategy based on our discussions with IEE analysts because each time a decision is made during configuration with customers, analysts would like to keep decisions in the PL diagram consistent and not to have multiple contradictions at a time. Tolerating contradictions during decision-making makes it hard to communicate with untrained customers for reasoning on decisions and resolving contradictions. For the resolution of decision contradictions in the PL use case diagram, our underlying assumption is that the diagram is configurable. A PL use case diagram is configurable if at least one valid PS use case diagram can be generated from the PL diagram. It may not be configurable because of an incorrect combination of variant and variation point dependencies (see Section 4.6.4).

4.6.2 Propositional Logic Mappings

Our consistency checking algorithm is based on mapping variation points, use cases and variant dependencies to propositional logic formulas. We assume that a PL use case diagram *PLD* is defined as a set, where each use case is a member of the set. The PL diagram consists of *n* use cases $PLD = \{u_1, ..., u_n\}$; each use case u_i in *PLD* is represented by a boolean variable with the same name. Boolean variable u_i evaluates to *true* if use case u_i is selected and *false* otherwise. If there is no decision made yet for use case u_i, variable u_i is not valued (*unknown*). Please note that all essential use cases are automatically selected. Figure 4.7 gives the corresponding propositional formulas for each pattern involving dependencies, variation points, and variant use cases, where propositions capture logical relationships among variant use cases. For instance, according to the corresponding propositional formula in Fig. 4.7(a), if use case UCA_m is selected for a product then the selection logically implies that use case UCB_n is also selected. Fig. 4.7(c) depicts the mapping when there is a *require* dependency between two variation points *A* and *B*. In such a case, if at least one of the variant use cases in variation point *A* ($UCA_1 \vee ... \vee UCA_m$) is selected, then at least one of the variant use cases in variation point *B* ($\rightarrow UCB_1 \vee ... \vee UCB_n$) should also be selected.

In a SAT solver based approach, for the entire PL use case diagram, one propositional formula can

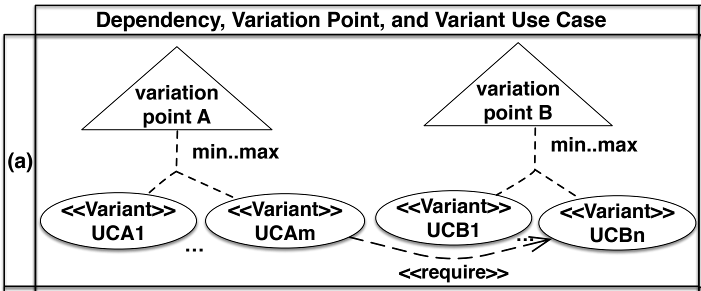
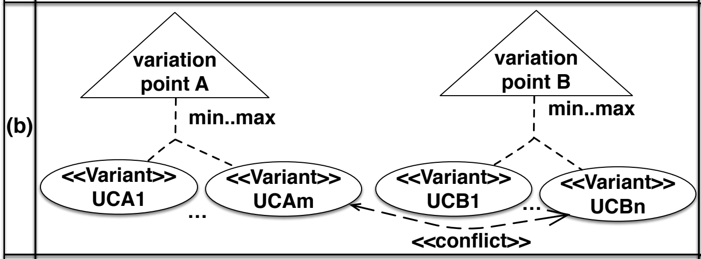
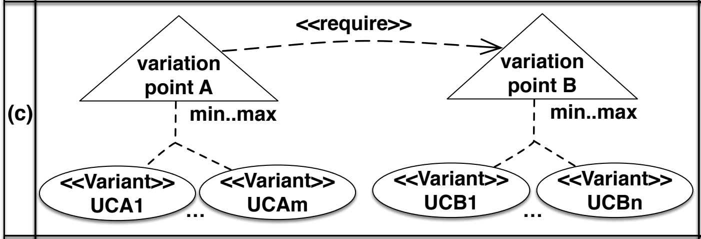
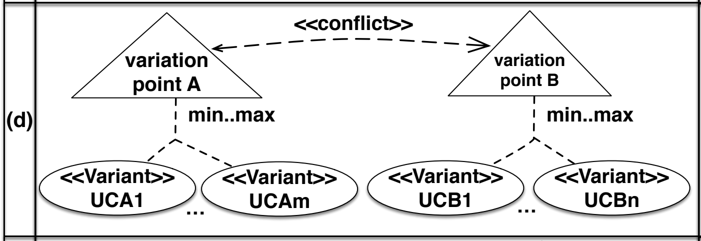
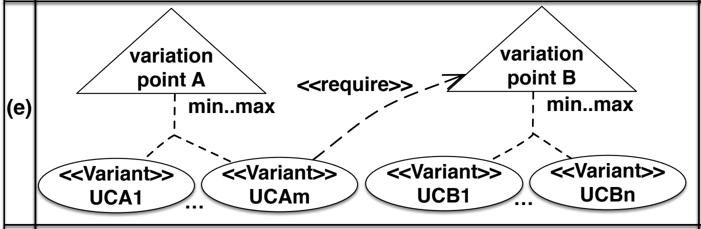
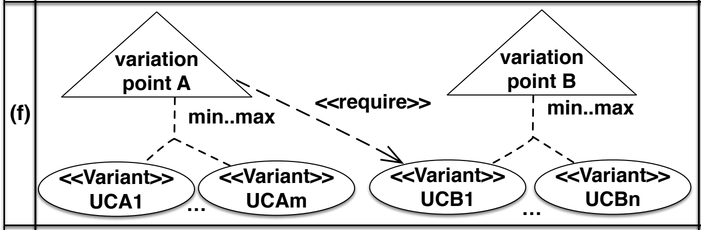
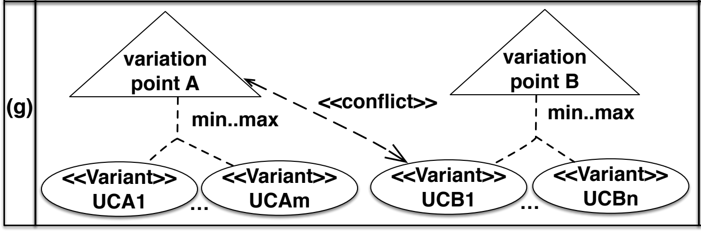
	Dependency, Variation Point, and Variant Use Case	Propositional Logic Mapping
(a)	 variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn <<require>>	$UCAm \rightarrow UCBn \quad (m \geqq 1 \text{ and } n \geqq 1)$
(b)	 variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn <<conflict>>	$\neg (UCAm \wedge UCBn) \quad (m \geqq 1 \text{ and } n \geqq 1)$
(c)	 <<require>> variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn	$(UCA1 \vee \ldots \vee UCAm) \rightarrow (UCB1 \vee \ldots \vee UCBn)$ $(m \geqq 1 \text{ and } n \geqq 1)$
(d)	 <<conflict>> variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn	$\neg ((UCA1 \vee \ldots \vee UCAm) \wedge (UCB1 \vee \ldots \vee UCBn))$ $(m \geqq 1 \text{ and } n \geqq 1)$
(e)	 variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm <<require>> variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn	$UCAm \rightarrow (UCB1 \vee \ldots \vee UCBn)$ $(m \geqq 1 \text{ and } n \geqq 1)$
(f)	 variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm <<require>> variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn	$(UCA1 \vee \ldots \vee UCAm) \rightarrow UCB1$ $(m \geqq 1 \text{ and } n \geqq 1)$
(g)	 variation point A min..max <<Variant>> UCA1 ... <<Variant>> UCAm <<conflict>> variation point B min..max <<Variant>> UCB1 ... <<Variant>> UCBn	$\neg ((UCA1 \vee \ldots \vee UCAm) \wedge UCB1)$ $(m \geqq 1 \text{ and } n \geqq 1)$

Figure 4.7. Mapping from PL Use Case Diagram to Propositional Logic

be formed as a conjunction of formulas derived from each dependency in the diagram using the mapping. Given such propositional formula and a set of variable assignments (decisions), a SAT solver can determine whether there is a value assignment to the remaining variables (undecided variation points) that will satisfy the predicate [Batory, 2005]. This approach may not be feasible for complex industrial projects when the number of variables to be computed is large. Therefore, to determine contradicting decisions, we follow an approach different than determining if there exists an interpretation that satisfies a propositional formula derived from the entire PL use case diagram (Section 4.6.3).

4.6.3 Consistency Checking Algorithm

For a given decision regarding a variation point in the PL diagram, our approach only checks the satisfaction of the propositional formulas derived from its dependencies. The number of variables taken into account in such approach is much smaller than the number of variables derived from the entire diagram to be computed by an approach using SAT solvers. Assume that we have two variant use cases *Ua* and *Ub* where *Ua requires Ub* and *Ua* is selected. The corresponding propositional formula in Fig. 4.7(a) is not satisfied only if *Ub* is unselected in prior decisions and there is no other further decision to be made for *Ub*. Therefore, we only check if *Ub* is unselected and cannot be selected in further decisions. If there is no decision made for *Ub* yet, we do not need to check if the corresponding formula is satisfied. The satisfaction of the formula is checked only if there is a valuation of the variable in the formula for Ub based on configuration decisions. However, decisions for other variant use cases might imply a decision for *Ub*. Our approach automatically infers those implicit decisions to be taken into account in the valuation of formulas.

Alg. 1 describes the part of our configuration algorithm related to consistency checking. For each new decision made by the analyst, the algorithm checks if the formulas derived for the decided use case elements are satisfied. If the formulas are not satisfied, the algorithm returns contradicting decisions to the analyst. The analyst updates the decisions to resolve the contradiction. In order to illustrate the algorithm, we rely on the example contradicting decisions in Fig. 4.3.

The analyst makes a decision *d* for each variation point *vp*, which is either included by an essential use case *uc* or not included by any use case (Lines 2, 4 and 5 in Alg. 1). For each new decision *d*, the algorithm checks if there is any contradicting, prior decision. Decision *d* is a quadruple of variation point *vp*, essential use case *uc* including *vp*, set of selected variant use cases *SUC* in *vp*, and set of unselected variant use cases *NSUC* in *vp* (Line 8). For the decisions in *VP1* and *VP4* in Fig. 4.3, *d* is $(VP1, UC1, \{UC2, UC4\}, \{UC3\})$ and $(VP4, \mathit{null}, \{UC9\}, \emptyset)$, respectively. The algorithm first determines if *d* complies with the cardinality constraint in *vp* (Line 9). If the cardinality constraint is satisfied, the algorithm checks if *d* contradicts any prior decision (Lines 10-27); otherwise, the analyst is asked to update decision *d* for the cardinality constraint (Line 29). We call some check functions (i.e., *checkConflictingVP*, *checkRequiringVP*, *checkRequiredVP*, *checkConflictingUC*, *checkRequiringUC*,

Alg. 1: Part of config

```
   Inputs : PL use case diagram, PLD, Set of PL use case specifications, PLS,
            PL domain model, PLDM
   Output: Set of PS use case models
1  Let DC be the empty set for completed decisions;
2  Let L be the set of pairs of variation points vp and use cases uc such that use cases are essential
   and they include the variation points, or the variation points are not included by any use case;
3  $T \leftarrow L$;
4  while $L \neq \emptyset$ do
5  |   $dp \in L$;
6  |   Let SUC be the set of variant use cases selected in $dp.vp$;
7  |   Let NSUC be the set of variant use cases unselected in $dp.vp$;
8  |   Let d be the quadruple $(dp.vp, dp.uc, SUC, NSUC)$;
9  |   if (d satisfies cardinality constraints in dp.vp) then
10 |   |   Let C be the empty set for contradicting decisions;
11 |   |   $C \leftarrow$ checkConflictingVP($dp.vp$, $DC$, $d$, $PLD$);
12 |   |   $C \leftarrow C \cup$ checkRequiringVP($dp.vp$,$DC$,$d$,$PLD$);
13 |   |   $C \leftarrow C \cup$ checkRequiredVP($dp.vp$,$DC$,$d$,$PLD$);
14 |   |   foreach ($u \in SUC$) do
15 |   |   |   $C \leftarrow C \cup$ checkConflictingUC($u$,$DC$,$d$,$PLD$);
16 |   |   |   $C \leftarrow C \cup$ checkRequiringUC($u$,$DC$,$d$,$PLD$);
17 |   |   end foreach
18 |   |   foreach ($u \in NSUC$) do
19 |   |   |   $C \leftarrow C \cup$ checkRequiredUC($u$, $DC$, $d$, $PLD$);
20 |   |   end foreach
21 |   |   if ($C = \emptyset$) then
22 |   |   |   $DC \leftarrow DC \cup \{d\}$;     $L \leftarrow L \setminus \{dp\}$;
23 |   |   |   Let $new_p = \{(vp, uc) \mid uc$ includes $vp \wedge uc \in SUC \wedge (vp, uc) \notin T\}$;
24 |   |   |   $L \leftarrow L \cup new_p$;          $T \leftarrow T \cup new_p$;
25 |   |   else
26 |   |   |   updateDecisions($C \cup \{d\}$);
27 |   |   end if
28 |   else
29 |   |   updateDecisions($\{d\}$);
30 |   end if
31 end while
32 ...
```

and *checkRequiredUC*) to determine whether the propositional logic formulas, derived from the dependencies to/from the diagram elements decided in *d*, are satisfied by *d* and set of prior decisions *DC* (Lines 10-20). If there is a formula not satisfied, there is at least one prior decision that is contradicting *d*. The algorithm reports contradicting decisions to be updated by the analyst in the *updateDecisions* function (Line 26). If there is no contradicting decision, *d* is approved and considered *completed* (Lines 21-22). The selected variant use cases may include variation points. The pairs of those variation points and their including use cases are considered for further decisions (Lines 23-24). Each *check* function in Alg. 1 checks the propositional formulas in one or more mappings in Fig. 4.7.

- ***checkConflictingVP*** checks the formulas for selected variation point vp conflicting with a variation point or a variant use case (Fig. 4.7(d) and (g)),
- ***checkConflictingUC*** checks the formulas for selected variant use case u conflicting with a variation point or a variant use case (Fig. 4.7(b) and (g)),
- ***checkRequiringVP*** checks the formulas for selected variation point vp requiring a variation point or a variant use case (Fig. 4.7(c) and (f)),
- ***checkRequiredVP*** checks the formulas for unselected variation point vp required by a variation point or a variant use case (Fig. 4.7(c) and (e)),
- ***checkRequiringUC*** checks the formulas for selected variant use case u requiring a variation point or a variant use case (Fig. 4.7(a) and (e)),
- ***checkRequiredUC*** checks the formulas for unselected variant use case u required by a variation point or a variant use case (Fig. 4.7(a) and (f)).

In Fig. 4.3, the decision in $VP1$ contradicts the prior decision in $VP4$ because of the *require* dependency. This is determined by the function *checkRequiredUC*, whose algorithm is presented in Alg. 2. For the rest of the *check* functions, the reader is referred to Supplementary Material.

Alg. 2 checks the formulas in Fig. 4.7(a) and (f) for an unselected variant use case required by a variation point or a variant use case. For instance, in Fig. 4.7(a), when *UCBn* is unselected and there is no further decision made for *UCBn*, it checks if *UCAm* is selected in any prior decision. If *UCAm* is already selected, it reports a contradiction.

Alg. 2 takes as input use case *uc* unselected in decision d, set of prior decisions *DC*, decision d, and PL use case diagram *PLD*, while it returns the set of decisions contradicting d. The inputs of *checkRequiredUC* for the example in Fig. 4.3 are $UC3$, $\{D1\}$, $D2$, and the PL diagram in Fig. 4.3 where $D1 = (VP4, null, \{UC9\}, \emptyset)$ and $D2 = (VP1, UC1, \{UC2, UC4\}, \{UC3\})$. The functions used in Alg. 2 are the following:

- ***inferUnselectedElements*** infers unselected elements for use case *uc* unselected in decision d (Line 4),
- ***inferSelectedElements*** infers selected elements for use case u selected in decision *dm* (Line 13),
- ***getRequiredElements*** returns use cases and variation points required by other variation points

Alg. 2: checkReqiredUC

```
   Input  : Use case, $uc$, Set of decisions, $DC$,
            Decision, $d$, PL use case diagram, $PLD$
   Output: Set of contradicting decisions
1  Let $RES$ be the empty set for contradicting decisions;
2  if ((uc is not selected in the completed decisions) and (there is no further decision to be made
   for uc)) then
3      Let $EX$ be the empty set for inferred, unselected elements;
4      $EX \leftarrow \{uc\} \cup$ inferUnselectedElements$(uc, d, DC, \emptyset, PLD)$;
5      foreach $(dm \in DC)$ do
6          Let $SUC$ be the set of selected use cases in $dm$;
7          Let $vp$ be the variation point in $dm$;
8          if $(SUC \neq \emptyset)$ and $((EX \cap$ getRequiredElements$(vp, PLD)) \neq \emptyset)$ then
9              $RES \leftarrow RES \cup \{dm\}$;
10         end if
11         foreach $(u \in SUC)$ do
12             Let $I$ be the empty set for inferred, selected elements;
13             $I \leftarrow \{u\} \cup$ inferSelectedElements$(u, dm, DC \cup \{d\}, \emptyset, PLD)$;
14             foreach $(e \in I)$ do
15                 if $(EX \cap$ getRequiredElements$(e, PLD) \neq \emptyset)$ then
16                     $RES \leftarrow RES \cup$ getInvolvedDecisions$(e, dm, d, DC)$;
17                 end if
18             end foreach
19         end foreach
20     end foreach
21     return $RES$;
22 else
23     return $RES$;
24 end if
```

or variant use cases (vp in Line 8 and e in Line 15),

- ***getInvolvedDecisions*** returns all decisions contradicting decision d for element e in decision dm (Line 16).

Alg. 2 starts with checking whether *uc*, unselected in decision d, is also unselected in the set of prior decisions *DC*, while it is also not possible to make any further decision for *uc* (Line 2). If *uc* is already selected in another decision or if there is still yet another decision to be made for *uc*, the function returns an empty set of contradicting decisions (Line 23); otherwise, the function checks whether there is any selected use case which requires *uc* (Lines 3-21). For $UC3$, the decision can be made only via the pair ($VP1$, $UC1$) since $UC1$ is the only use case including $VP1$. $D2$ is the decision made via the pair ($VP1$, $UC1$) where $UC3$ is unselected.

When *uc* is unselected in *d*, there might be other variant use cases automatically unselected. These use cases are in the variation points included by *uc* (Line 4). The function $inferUnselectedElements$ returns the inferred, unselected use cases and variation points for use case *uc* unselected in decision d. For $D2$, it infers $UC7$, $UC8$, and $VP3$ ($EX = \{UC3, UC7, UC8, VP3\}$). $VP3$ is included only by $UC3$. $UC7$ and $UC8$ in $VP3$ cannot be selected after $UC3$ is unselected in $D2$. Since all variant use cases in $VP3$ are automatically unselected, $VP3$ is also considered *unselected*. If any of these elements in the set of unselected elements *EX* is required by a variation point selected in prior decision *dm* in *DC*, the current decision *d* contradicts *dm* (Lines 6-10). The variant use cases selected in *dm* (*SUC* in Line 6) might cause other variant use cases automatically to be selected. These are the use cases with a mandatory variability relation in the variation points included by the use cases selected in *dm* (Line 13). The function *inferSelectedElements* infers those variant use cases to check whether they require any unselected element in *EX* (Lines 11-19). For $D1$, there is no inferred variant use case. There is only $UC9$ which is selected for $VP4$ in $D1$ ($I = \{UC9\}$ for $u = UC9$). Only $UC9$, selected in $D1$, requires $UC8$ in *EX* (Line 15). There might be other prior decisions contributing to the selection of $UC9$. The function *getInvolvedDecisions* returns all these decisions (Line 16). There is only $D1$ in which $UC9$ is selected. Therefore, the function *checkRequiredUC* returns only $D1$ in the set of contradicting decisions ($RES = \{D1\}$ in Lines 16 and 21).

4.6.4 Non-configurable PL Use Case Diagrams

As stated in Section 4.6.1, our assumption for consistency checking is that the PL diagram is configurable. Fig. 4.8 provides an example of a non-configurable PL diagram.

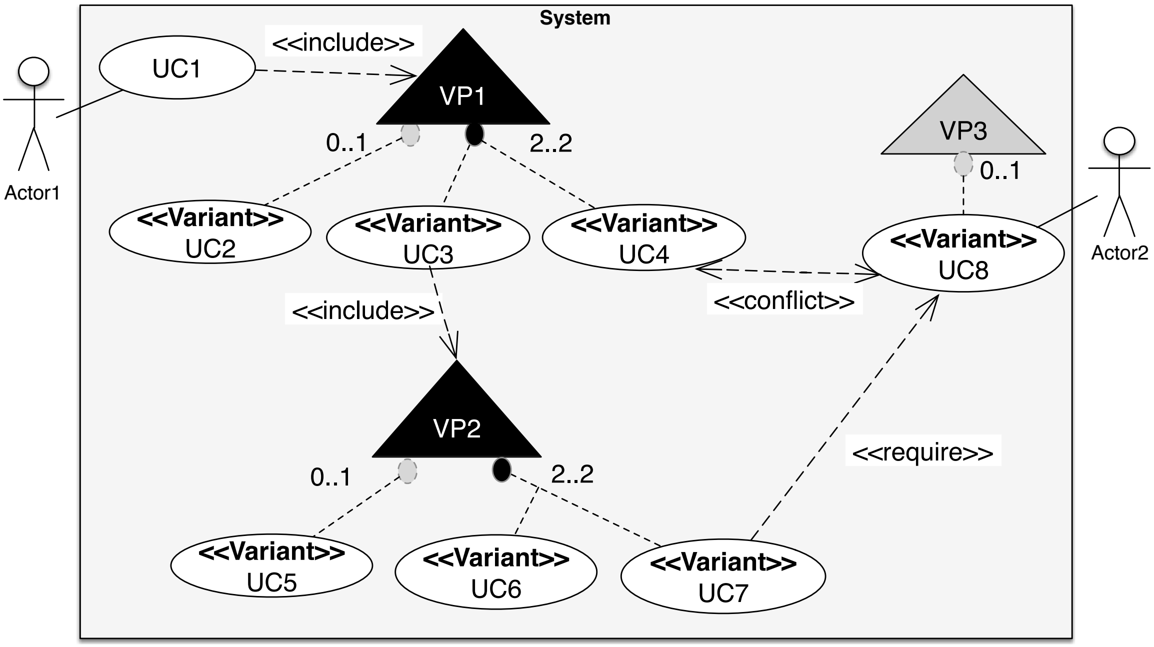


Figure 4.8. An Example of a Non-Configurable PL Use Case Diagram

Essential use case *UC1* includes mandatory variation point *VP1* which has three variant use cases through optional and mandatory variability relations. Variant use cases *UC3* and *UC4* in *VP1* are always selected because of the mandatory variability relation (cardinality constraint '*2..2*' in *VP1*). *UC3* includes another mandatory variation point, *VP2*, which has three variant use cases *UC5*, *UC6* and *UC7*. *UC6* and *UC7* are always selected because of the mandatory variability relation (cardinality constraint '*2..2*' in *VP2*). Therefore, variant use cases *UC3*, *UC4*, *UC6*, and *UC7* are automatically selected for every product. *UC7 requires* another variant use case, *UC8*, while *UC8 conflicts* with *UC4*. The *require* dependency implies that if *UC7* is selected for a product, *UC8* should also be selected for the same product. Since *UC7* is automatically selected for every product, *UC8* should always be selected for every product. On the other hand, *UC8* cannot be selected for any product because it conflicts with *UC4*, which is also automatically selected for every product. Therefore, it is not possible to generate a valid PS use case diagram from the PL diagram in Fig. 4.8. The combination of the dependencies *require* between *UC8* and *UC7*, *conflict* between *UC8* and *UC4*, and *include* between *UC3* and *VP2* and the cardinality constraints (*2..2*) in *VP1* and *VP2* is the reason of non-configurability in Fig. 4.8. Such combination of dependencies and cardinality constraints in a non-configurable PL use case diagram needs to be resolved before making configuration decisions. Otherwise, in the non-configurable PL diagram, there will always be contradicting decisions which are impossible to resolve.

The detection of non-configurable models has been addressed in the context of feature modeling and product line requirements specifications [Benavides et al., 2010, Trinidad and Ruiz-Cortés, 2009, Trinidad et al., 2008]. There are also techniques [Goknil et al., 2011, Goknil et al., 2008a, Goknil et al., 2013] to identify incorrect combinations of requirements dependencies in a broader context. Existing techniques (e.g., [Rosa et al., 2009, Goknil et al., 2011, Sun et al., 2005, Wang et al., 2005, Lauenroth and Pohl, 2007, Lauenroth and Pohl, 2008, Stoiber, 2012, Durán et al., 2016]) could be adapted for PL use case diagrams in our configuration approach as part of Step 1, *Elicit Product Line Use Case and Domain Models*, in Fig. 4.1. Before making decisions, the analyst could automatically check if the PL use case diagram is configurable and, if necessary, could resolve the incorrect combination of dependencies and cardinality constraints.

4.7 Generation of Product Specific Use Case Models

After the decisions are made, the PS use case and domain models are generated from the PL models. The generation of PS models are implemented in the *PL-PS Transformer* component in *PUMConf* (Fig. 4.9). The details of the component architecture of *PUMConf* are given in Section 4.8.

The *PL-PS Transformer* contains three subcomponents: *Diagram Transformer*, *Specification Transformer* and *Domain Model Transformer*. The subcomponents are *update-in-place* transformations [Czarnecki and Helsen, 2006] in which a model is both an input and output. Each subcomponent takes one of the PL models and relevant decisions in the decision model as input, and produces the correspond-

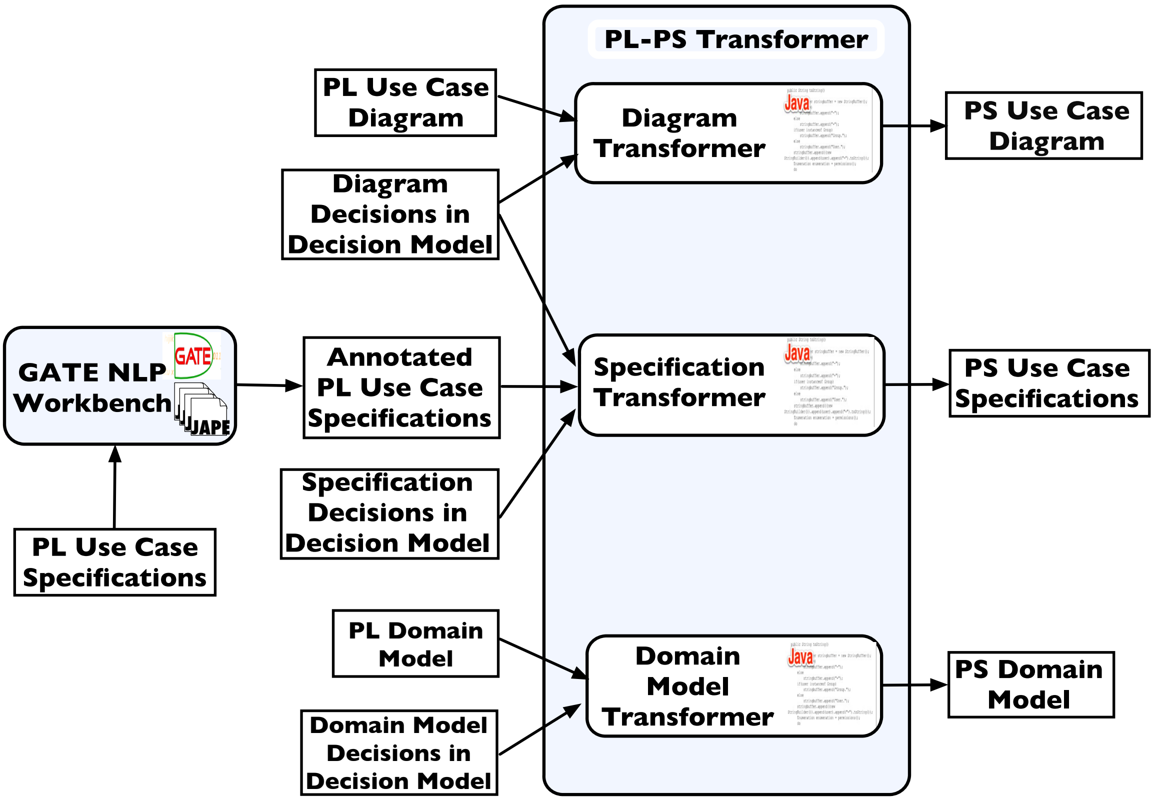

PL-PS Transformer
PL Use Case Diagram
Diagram Decisions in Decision Model
Diagram Transformer
PS Use Case Diagram
GATE NLP Workbench
GATE
JAPE
Annotated PL Use Case Specifications
Specification Transformer
PS Use Case Specifications
PL Use Case Specifications
Specification Decisions in Decision Model
PL Domain Model
Domain Model Decisions in Decision Model
Domain Model Transformer
PS Domain Model
Java

Figure 4.9. Overview of the PS Use Case and Domain Model Generation

ing PS model as output.

All PL and PS use case specifications are stored as plain text in the native IBM DOORS format. During the consistency checking of use case and domain models (see Step 2 in Fig. 4.1), for the RUCM keywords and types of steps, use case specifications in IBM DOORS are annotated using NLP provided by the GATE workbench (see Section 4.8). The *Specification Transformer* uses the annotations to distinguish RUCM steps and types of alternative flows in matching transformation rules (Section 4.8). It processes the plain text instead of instances of the RUCM metamodel [Zhang et al., 2013]. Compared to model transformation languages, Java provides much more flexibility for handling annotated plain text in terms of loading, matching and editing the text. Therefore, we used Java to implement the *Specification Transformer*. To provide the uniformity in the *PL-PS Transformer*, other subcomponents are also implemented in Java.

The *Diagram Transformer* takes the PL use case diagram and diagram decisions as input, and generates the PS use case diagram as output. The PS diagram generation is based on mappings between patterns in PL and PS diagrams driven by decisions. For instance, for a variant use case in the PL diagram, there is a corresponding use case in the PS diagram only if the variant use case is selected during decision-making. Fig. 4.10 gives example source and target patterns with decisions for use case diagrams.

Fig. 4.10 has three columns, i.e., *source pattern*, *decision*, and *target pattern*, to represent example

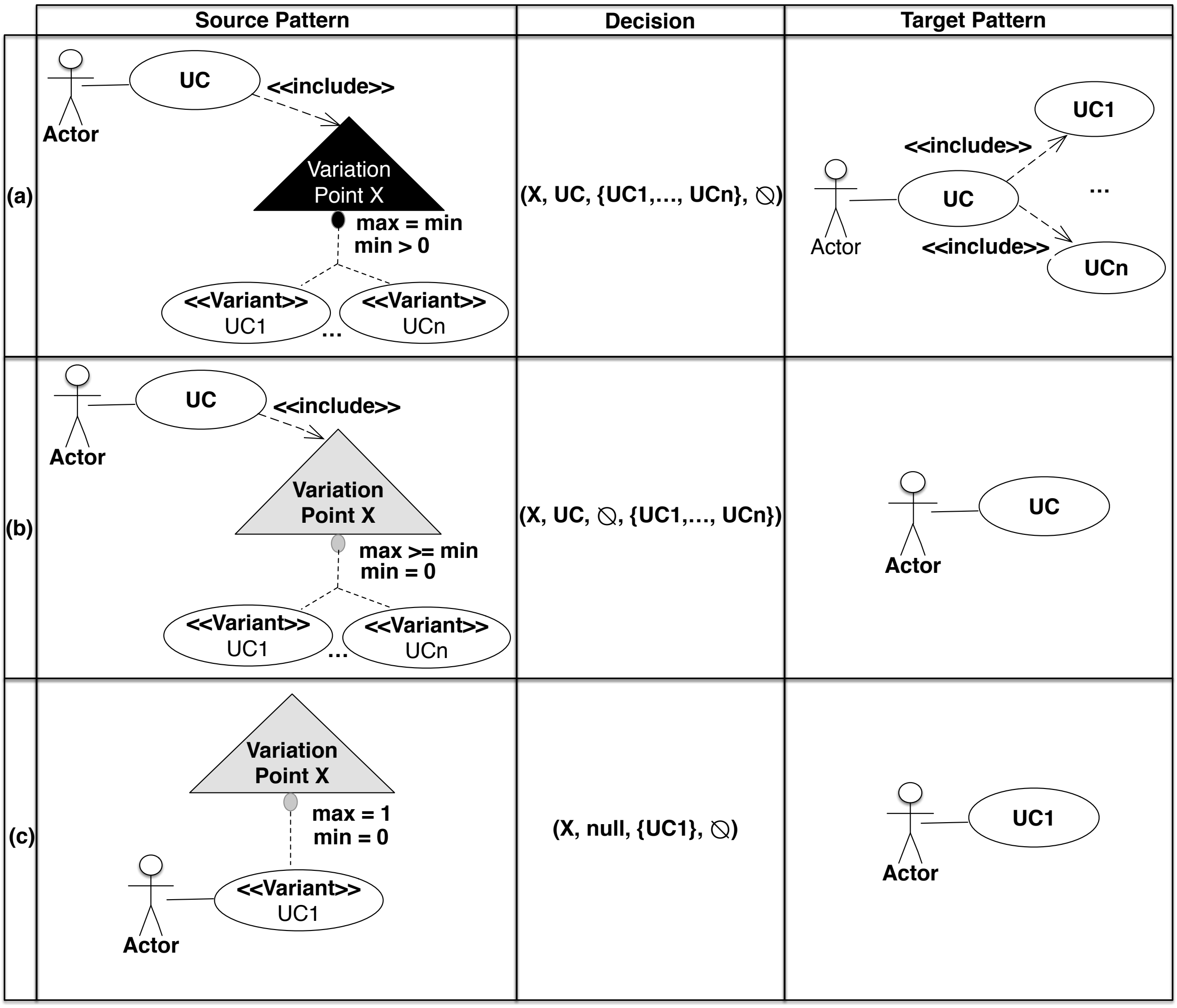
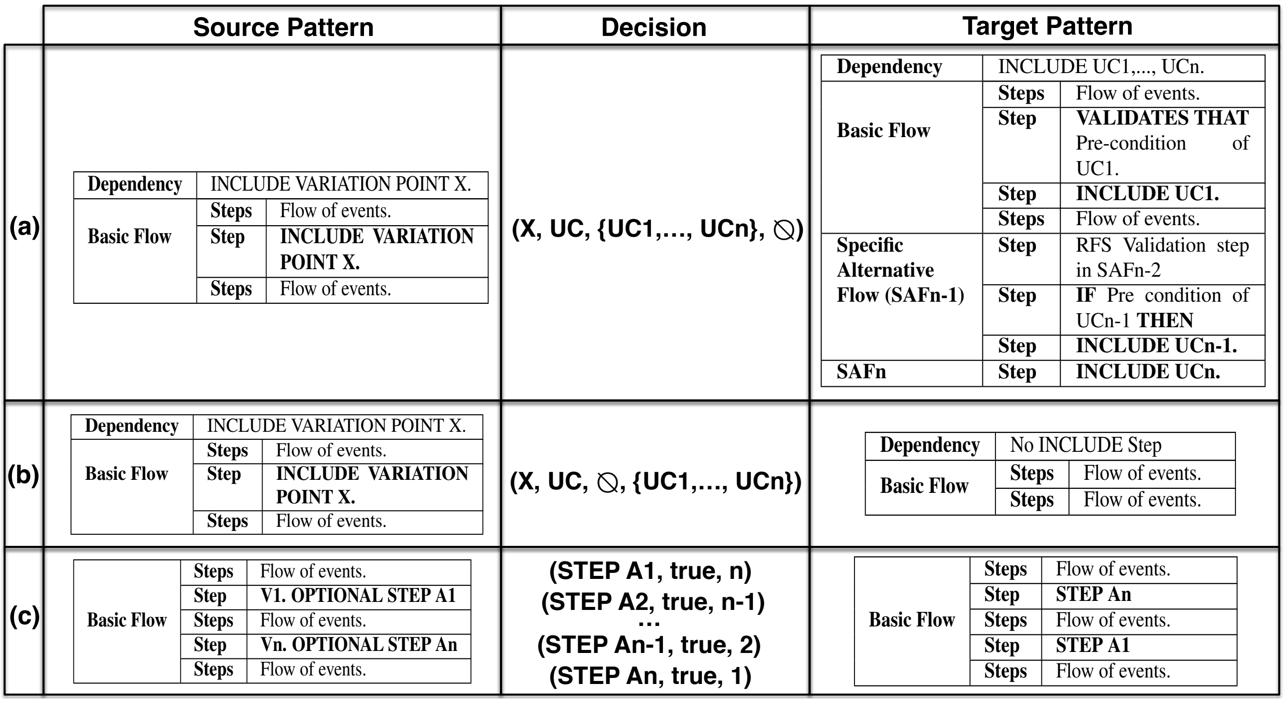


Figure 4.10. Example Source and Target Patterns with Decisions for Use Case Diagrams

mappings between PL and PS diagrams. Decisions for the diagrams are represented as quadruples (see Section 4.6). In Fig. 4.10(a), the analyst selects all variant use cases in mandatory variation point *X* included by essential use case *UC*. The variation point, *include* relation and unselected variant use cases are removed while each selected variant use case is transformed into a use case included by *UC* in the PS diagram. In Fig. 4.10(b), optional variation point *X* is included by essential use case *UC* while all variant use cases are unselected. Only essential use case *X* is kept in the PS diagram. The source pattern in Fig. 4.10(c) represents optional variation point *X* which has one variant use case (i.e., *UC*1). *X* is not included by any other essential or variant use case. When *UC*1 is selected, only variation point *X* is removed from the PS diagram.

The *Specification Transformer* takes both diagram and specification decisions to generate PS use case specifications (see Fig. 4.11 for example mappings between PL and PS specifications). Diagram decisions are used to transform use case steps where variation points are included. In Fig. 4.11(a), the source pattern represents an example specification where the variation point *X* in Fig. 4.10(a) is included in between other use case steps. For the same decision, represented as a quadruple, in



	Source Pattern	Decision	Target Pattern
(a)	**Dependency** INCLUDE VARIATION POINT X. **Basic Flow** **Steps** Flow of events. **Step** **INCLUDE VARIATION POINT X.** **Steps** Flow of events.	(X, UC, {UC1,..., UCn}, ⦸)	**Dependency** INCLUDE UC1,..., UCn. **Basic Flow** **Steps** Flow of events. **Step** **VALIDATES THAT** Pre-condition of UC1. **Step** **INCLUDE UC1.** **Steps** Flow of events. **Specific Alternative Flow (SAFn-1)** **Step** RFS Validation step in SAFn-2 **Step** **IF** Pre condition of UCn-1 **THEN** **Step** **INCLUDE UCn-1.** **SAFn** **Step** **INCLUDE UCn.**
(b)	**Dependency** INCLUDE VARIATION POINT X. **Basic Flow** **Steps** Flow of events. **Step** **INCLUDE VARIATION POINT X.** **Steps** Flow of events.	(X, UC, ⦸, {UC1,..., UCn})	**Dependency** No INCLUDE Step **Basic Flow** **Steps** Flow of events. **Steps** Flow of events.
(c)	**Basic Flow** **Steps** Flow of events. **Step** **V1. OPTIONAL STEP A1** **Steps** Flow of events. **Step** **Vn. OPTIONAL STEP An** **Steps** Flow of events.	(STEP A1, true, n) (STEP A2, true, n-1) ... (STEP An-1, true, 2) (STEP An, true, 1)	**Basic Flow** **Steps** Flow of events. **Step** **STEP An** **Steps** Flow of events. **Step** **STEP A1** **Steps** Flow of events.

Figure 4.11. Example Source and Target Patterns with Decisions for Use Case Specifications

Fig. 4.10(a) where all variant use cases are selected, the *VALIDATES THAT* and *INCLUDE* steps in the basic flow and a set of alternative flows are generated for the PS specification. One of the variant use cases ($UC1$) is executed in the basic flow (the step where $UC1$ is included) if its precondition holds (the *VALIDATES THAT* step). The rest of the selected use cases are executed in the generated alternative flows. One of the alternative flows is taken if the *VALIDATES THAT* statement in the basic flow fails. In Fig. 4.11(b), the source PL specification for Fig. 4.10(b) is transformed into the PS specification based on the diagram decision where no variant use case is selected. The step where the variation point X is included is removed for the PS specification.

Fig. 4.11(c) represents an example source pattern which contains multiple optional steps in a variant order. The analyst has to decide which optional steps are retained and in which order in the PS specification. The specification decision for each optional step is represented as a triple of the step name, a boolean variable which is true when the step is selected, and the decided order number. In the example, each optional step is selected in a reverse order for the PS specification (see the decision column in Fig. 4.11(c)). Based on the decisions, the selected optional steps (*OPTIONAL STEP* A_1, ..., *OPTIONAL STEP* A_{n-1}, *OPTIONAL STEP* A_n) are preserved in the reverse order (*STEP* A_n, *STEP* A_{n-1}, ... , *STEP* A_1) in the target pattern while there might be common steps in between the selected optional steps.

The *Domain Model Transformer* takes the PL domain model and domain model decisions as input to generate a PS domain model. As we discussed in Section 4.5, the generation of a PS domain model is straightforward: in addition to mandatory entities, all selected optional and variant domain entities are kept, while their PL stereotypes are removed. The *is-a* relations having only a single subclass are also removed to prune the generated model. In the next section, we present the tool support and architecture, which provide more detailed information about the interaction of the *PL-PS Transformer* with other components of the tool.

4.8 Tool Support

We have implemented our configuration approach in a prototype tool, *PUMConf (Product line Use case Model Configurator)*. Section 4.8.1 provides the layered architecture of the tool while we describe the tool features with some screenshots in Section 4.8.2. For accessing the tool executables, see: `https://sites.google.com/site/pumconf/`.

4.8.1 Tool Architecture

The tool architecture is composed of three layers (see Fig. 4.12): (i) the *User Interface (UI) layer*, (ii) the *Application layer*, and (iii) the *Data layer*.

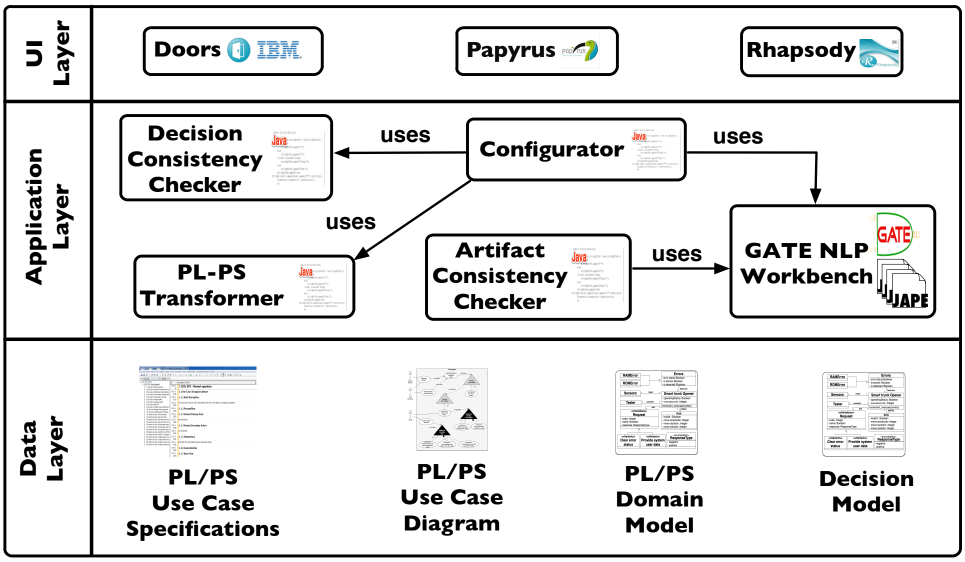


Figure 4.12. Layered Architecture of PUMConf

User Interface (UI) Layer. This layer supports the activity of eliciting product line use cases and domain models to create or update the PL artifacts (see Fig. 4.1). It also enables the viewing of the generated PS artifacts. We employ IBM Doors (`www.ibm.com/software/products/ca/en/ratidoor/`) for use case specifications, Papyrus (`https://www.eclipse.org/papyrus/`) for use case diagrams, and IBM Rhapsody (`www.ibm.com/software/products/en/ratirhapfami`) for domain models. IBM Doors does not put any restriction on the structure of use cases and thus allows the adoption of the RUCM template. Halmans and Pohl [Halmans and Pohl, 2003] do not provide any metamodel or UML profile for the PL use case diagram extensions. Therefore, we implemented our own UML profile in Papyrus to enable the use of the Papyrus model editor for creating and editing PL use case diagrams. Our choice of IBM Rhapsody for domain models is based on the modeling practice at IEE, which has been based on Rhapsody. For domain modeling, we could also employ the Papyrus model editor which we use to present the decision model.

Application Layer. This layer supports the main activities of our approach in Fig. 4.1: *checking consistency of PL use case and domain models* and *configuring PS use case and domain models*. It contains four main components implemented in Java: *Configurator*, *Artifact Consistency Checker*, *Decision Consistency Checker*, and *PL-PS Transformer*. To access these *Application Layer* components through the *UI Layer*, we implemented an IBM DOORS plugin (see Fig. 4.13).

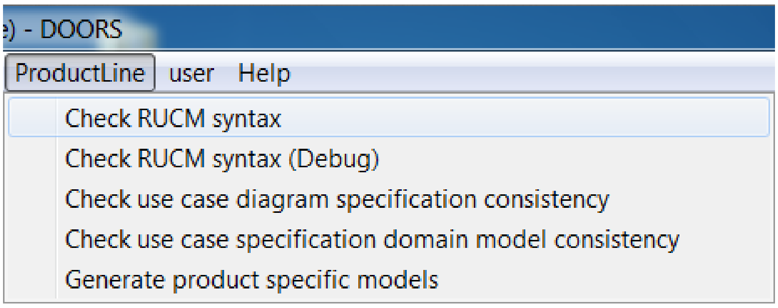

) - DOORS
ProductLine
user
Help
Check RUCM syntax
Check RUCM syntax (Debug)
Check use case diagram specification consistency
Check use case specification domain model consistency
Generate product specific models

Figure 4.13. Menu to Activate IBM DOORS Plug-ins for PUMConf

The *Configurator* component is a coordinator that manages two other components, i.e., *Decision Consistency Checker* and *PL-PS Transformer*. In addition to the *Configurator*, the user has direct access, via the DOORS plug-in, to the *Artifact Consistency Checker* which employs NLP to check the consistency of the PL use case diagram, the use case specifications complying with the RUCM template, and the domain model. The *Decision Consistency Checker* implements the parts of the configuration algorithm where a decision for the PL use case diagram is received from the analyst and its consistency with previous decisions is checked. The algorithm does not attempt to find a valid configuration but simply traverses the diagram for decisions to recursively check whether the implications of any variation point - variant use case dependency (i.e., *include*, *require*, and *conflict*) are violated by the configuration decisions (see Section 4.6).

The *PL-PS Transformer* component is the Java implementation of the transformation rules for the use case diagram, specifications and domain model. Before the application of the transformation rules, use case specifications need to be annotated by using NLP (see Section 4.7).

To perform NLP in use case specifications, the *Configurator* and *Artifact Consistency Checker* components use a regular expression engine, called JAPE [H. Cunningham et al, 2018], in the GATE workbench (`http://gate.ac.uk/`), an open-source Natural Language Processing (NLP) framework. We implemented the extended RUCM restriction rules in JAPE. With NLP, use cases are first split into tokens. Second, Part-Of-Speech (POS) tags (i.e., *verb*, *noun*, and *pronoun*) are assigned to each token. By using the RUCM restriction rules implemented in JAPE, blocks of tokens are tagged to distinguish RUCM steps (i.e., *output*, *input*, *include*, and *internal operations*) and types of alternative flows (i.e., *specific*, *alternative*, and *global*). The output of the NLP contains the annotated use case steps. The *Configurator* passes the annotations to the *PL-PS Transformer* in order to match the transformation rules while the *Artifact Consistency Checker* processes these annotations with the use case diagram and domain model to generate the list of inconsistencies among the artifacts. For instance, the consistency of use case specifications and domain model is checked by comparing the use case specification entities identified by the NLP application with the entities in the domain model. For each domain entity identified through NLP, the *Artifact Consistency Checker* generates an entity name by removing all white spaces and putting all first letters following white spaces in capital. If the entity name does not appear either as class name or as an attribute name in the domain model, or if the entity name is only mentioned in the optional parts of use case specifications while it appears as a mandatory entity in the domain model, an inconsistency is reported. The consistency checking could be extended with syntactic and semantic similarity checking techniques [Arora et al., 2015a, Arora et al., 2015b] to tackle inconsistent naming conventions in the comparison.

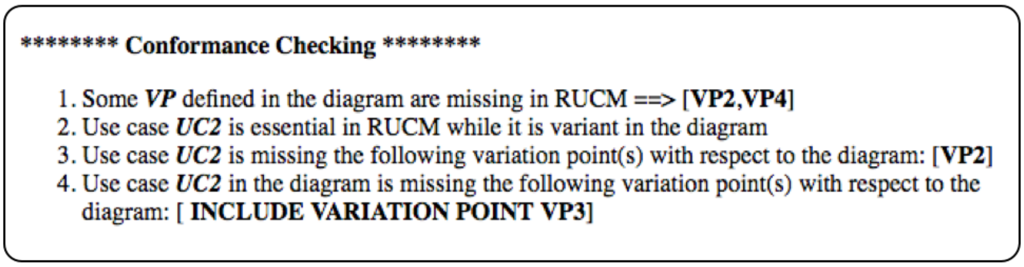
Data Layer. All the use case specifications are stored in the native IBM DOORS format while the domain model is exported into the XMI format by the Rhapsody XMI toolkit. The PL use case diagram and the generated PS diagram are stored using the UML profile mechanism, while the decision model is saved in Ecore [Ecl, 2018].

4.8.2 Tool Features

We describe the most important features of our tool: *managing PL use case and domain models*, *checking consistency of PL use case and domain models*, *getting configuration decisions from the analyst*, *checking consistency of decisions*, and *displaying decisions*. These features support the steps of the modeling process given in Fig. 4.1.

Managing PL use case and domain models. This feature supports Step 1, *Elicit Product Line Use Case and Domain Models*, in Fig. 4.1. The analyst can create, update, and delete the PL use case diagram, specifications, and domain model by using the selected modeling tools (i.e., IBM Doors, Papyrus, and Rhapsody) adopted in *PUMConf*.

Checking consistency of PL use case and domain models. The consistency of the PL use case and domain models needs to be ensured in Step 2, *Check Consistency of Product Line Use Case and Domain Models*, in Fig. 4.1 before the analyst makes decisions about the variability information. Our tool automatically checks (1) if the PL use case specifications conform to the RUCM template and its product line extensions, (2) if the PL use case diagram is consistent with the PL use case specifications, and (3) if the PL domain model is consistent with the PL use case specifications (see Fig. 4.13). Fig. 4.14 presents an example output of the consistency checking of the PL use case diagram and specifications in Section 4.5.



```
******** Conformance Checking ********

   1. Some VP defined in the diagram are missing in RUCM ==> [VP2,VP4]
   2. Use case UC2 is essential in RUCM while it is variant in the diagram
   3. Use case UC2 is missing the following variation point(s) with respect to the diagram: [VP2]
   4. Use case UC2 in the diagram is missing the following variation point(s) with respect to the
      diagram: [ INCLUDE VARIATION POINT VP3]
```

Figure 4.14. PUMConf User Interface for Reporting Inconsistencies

Three types of inconsistencies are reported in Fig. 4.14: missing variation points in the specifications, variant use cases given as essential ones in the corresponding specifications, and missing variation points in the diagram.

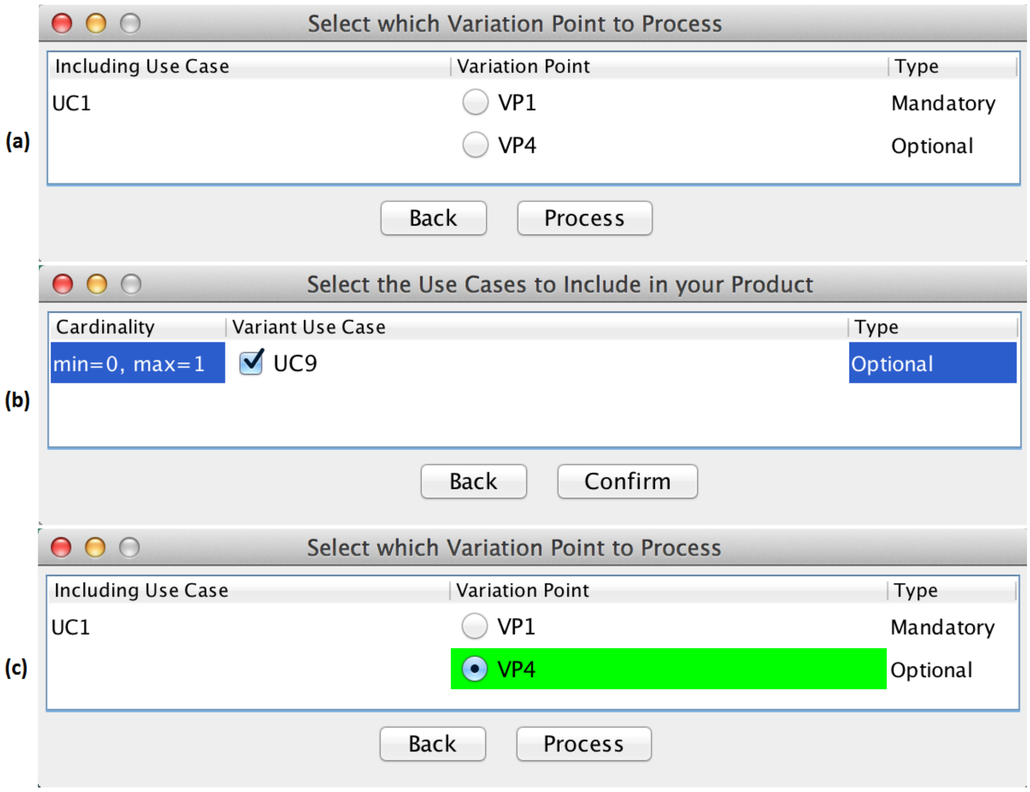
Getting configuration decisions from the analyst. During Step 3, *Configure Product Specific Use Case and Domain Models*, in Fig. 4.1, the tool first determines the list of variation points to be decided, based on the dependency structure of variation points, i.e., *include*. The analyst makes a decision for each variation point in the list, while the tool checks the consistency of the decision with prior decisions. A decision may cause further decisions to be made for some other variation points, i.e., included by the variant use cases selected in the decision. In such cases, after each decision the tool automatically updates the list of variation points to be decided. Fig. 4.15 presents the user interface for getting the decision for the variation point *VP4* in Fig. 4.3.

In Fig. 4.15(a), the tool lists the variation points *VP1* and *VP4* but not *VP2* and *VP3* since the analyst can make a decision for *VP2* and *VP3* only after *UC2* and *UC3* are selected in *VP1* (see Fig. 4.3). The analyst makes a decision in *VP4* by selecting *UC9* in Fig 4.15(b). After the decision is confirmed to be consistent with prior decisions, *VP4* is highlighted in green, indicating that the decision has been validated and recorded (see Fig. 4.15(c)).

After the decisions for the PL diagram are made, the analyst proceeds with the PL use case specifications and domain model. Fig. 4.16 presents the PUMConf's user interface for selecting optional steps in the PL specification of use case $UC2$ in Table 4.1.

In Fig. 4.16, the tool lists the entire use case specification including the *optional* tags on optional steps. The analyst makes a decision for each optional step and each optional alternative flow in the specification (none in $UC2$). After these decisions are made, the tool asks the order of the variant order steps if there are any in the specification.

Checking consistency of decisions. After each decision is made in the diagram, the tool checks its consistency with prior decisions. If there is any contradicting decision, the analyst is asked to update the current and/or previous decisions causing the contradiction.



Select which Variation Point to Process

Including Use Case	Variation Point	Type
UC1	○ VP1	Mandatory
	○ VP4	Optional

Select the Use Cases to Include in your Product

Cardinality	Variant Use Case	Type
min=0, max=1	☑ UC9	Optional

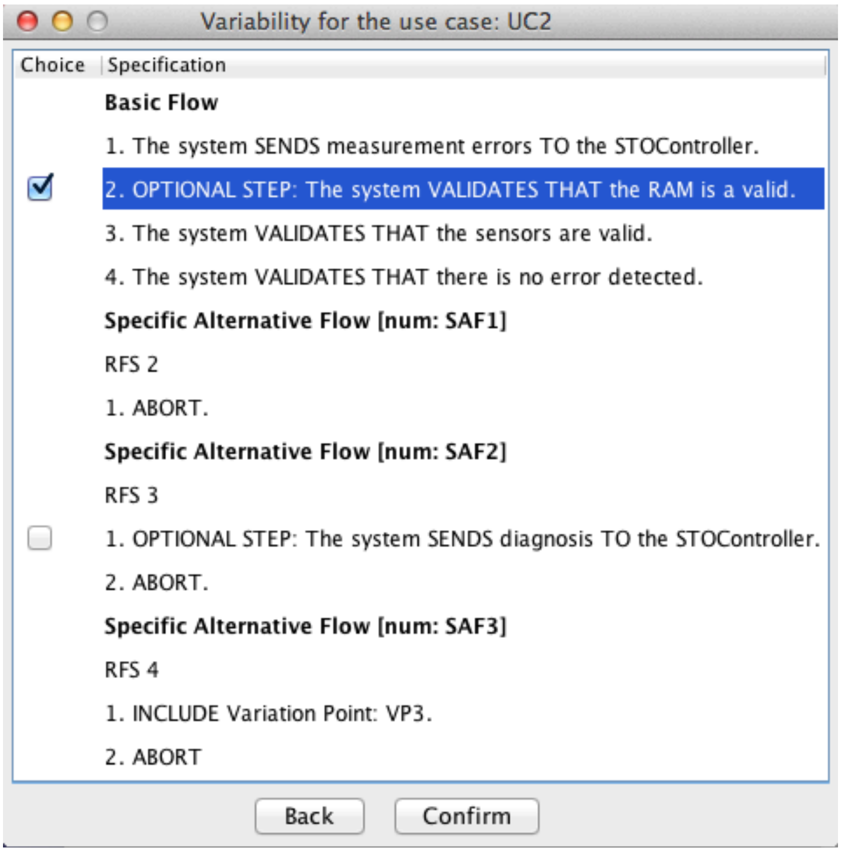
Select which Variation Point to Process

Including Use Case	Variation Point	Type
UC1	○ VP1	Mandatory
	● VP4	Optional

Figure 4.15. PUMConf's User Interface for (a) listing variation points driving decisions, (b) selecting variant use cases for the selected variation point, and (c) showing the updated list of variation points after the decision.

After the decision regarding *VP4* in Fig. 4.15(b), the analyst proceeds with *VP1* in Fig. 4.15(c) and then selects *UC2* and *UC4* (Fig. 4.17(a)). Please note that *UC2* is automatically selected since it is a mandatory variant use case implied by the cardinality constraint in *VP1*. When the analyst submits the decision, the tool automatically checks if it contradicts prior decisions. A contradiction for the decisions in *VP1* and *VP4* is reported (Fig 4.17(b)). The upper part of the user interface in Fig. 4.17(b) provides an explanation for the contradiction while the bottom part lists the decisions involved in the contradiction, with an *Edit* button to update the corresponding decision. To resolve the contradiction in Fig. 4.17(b), the analyst updates the decision in *VP1* by selecting *UC2* and *UC3* but not *UC4* (Fig. 4.17(c)). After the decision is updated, the tool checks again for contradictions. It confirms that there is no more contradiction and all decisions are consistent. *UC2* and *UC3* include *VP2* and *VP3*, respectively. Therefore, after *UC2* and *UC3* are selected, the pairs $(UC2, VP2)$ and $(UC3, VP3)$ are automatically given to the analyst to make further decisions (Fig. 4.17(d)).

Displaying decisions. After the configuration is completed with the generation of the PS use case and domain models, the analyst may need to reconfigure. The tool presents the entire set of decisions for the product with a user interface similar to Fig. 4.15(a) and (b).



Choice	Specification
	Basic Flow
	1. The system SENDS measurement errors TO the STOController.
☑	2. OPTIONAL STEP: The system VALIDATES THAT the RAM is a valid.
	3. The system VALIDATES THAT the sensors are valid.
	4. The system VALIDATES THAT there is no error detected.
	Specific Alternative Flow [num: SAF1]
	RFS 2
	1. ABORT.
	Specific Alternative Flow [num: SAF2]
	RFS 3
□	1. OPTIONAL STEP: The system SENDS diagnosis TO the STOController.
	2. ABORT.
	Specific Alternative Flow [num: SAF3]
	RFS 4
	1. INCLUDE Variation Point: VP3.
	2. ABORT

Figure 4.16. PUMConf's User Interface for Selecting Optional Steps in Use Cases.

4.9 Evaluation

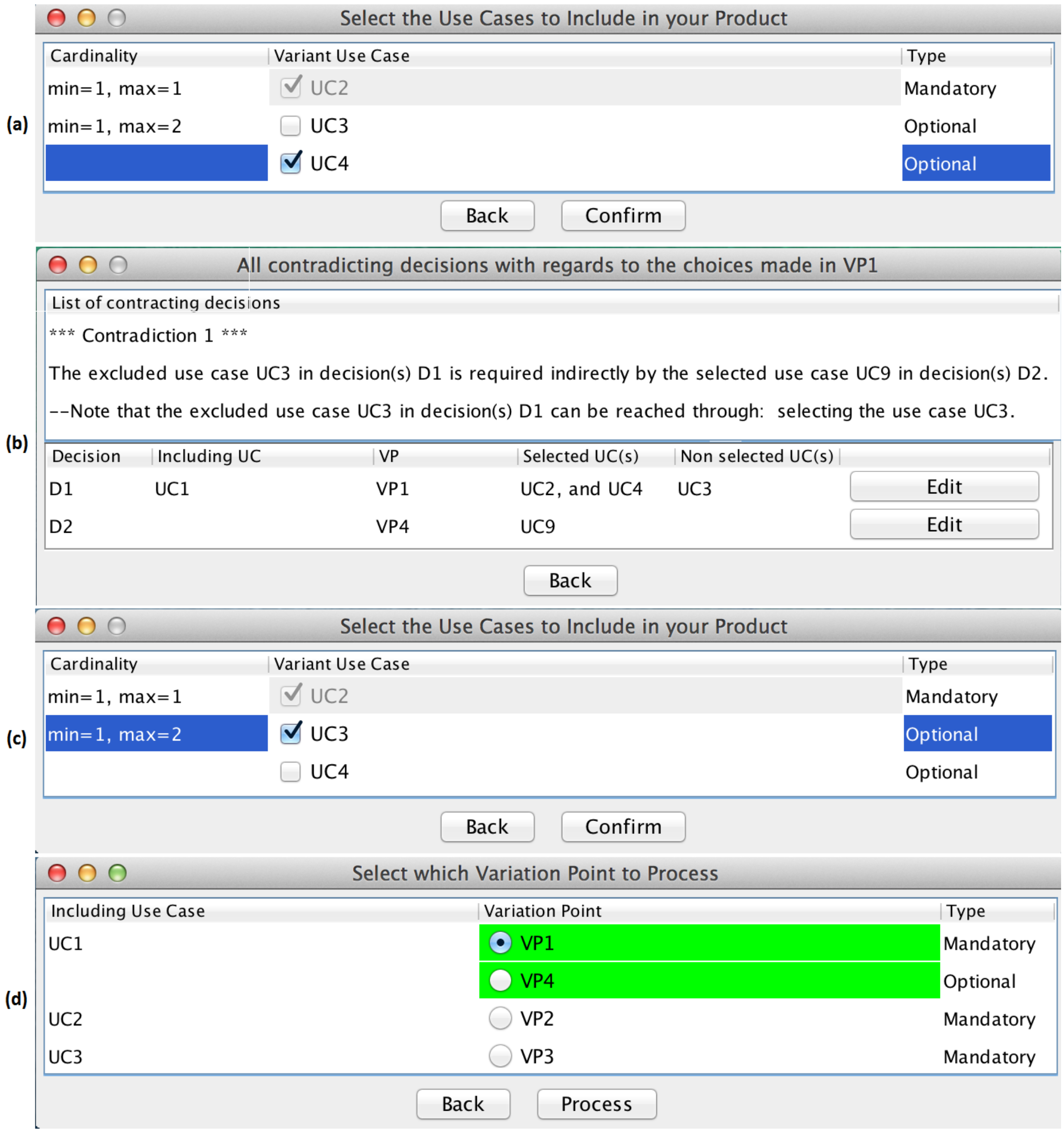
In this section, we evaluate our configuration approach via reporting (i) an industrial case study, i.e., STO, to demonstrate its feasibility (Section 4.9.1), (ii) the results of a questionnaire based survey at IEE aiming at investigating how PUMConf is perceived to address the challenges listed in Section 4.2 (Section 4.9.2), and (iii) discussions with the IEE analysts to gather more insights into the benefits and challenges of applying it in an industrial setting (Section 4.9.3).

4.9.1 Industrial Case Study

We report our findings about the feasibility of our approach and its tool support in an industrial context. In order to experiment with PUMConf in an industrial project, we applied it to the functional requirements of STO.

4.9.1.1 Goal

Our goal was to assess, in an industrial context, the feasibility of using PUMConf to improve variability modeling and reuse in the context of use case and domain models. STO was selected for this assessment since it was a relatively new project at IEE with multiple potential customers requiring different features.



(a) Select the Use Cases to Include in your Product

Cardinality	Variant Use Case	Type
min=1, max=1	☑ UC2	Mandatory
min=1, max=2	□ UC3	Optional
	☑ UC4	Optional

(b) All contradicting decisions with regards to the choices made in VP1

List of contracting decisions

*** Contradiction 1 ***

The excluded use case UC3 in decision(s) D1 is required indirectly by the selected use case UC9 in decision(s) D2.

--Note that the excluded use case UC3 in decision(s) D1 can be reached through: selecting the use case UC3.

Decision	Including UC	VP	Selected UC(s)	Non selected UC(s)	
D1	UC1	VP1	UC2, and UC4	UC3	Edit
D2		VP4	UC9		Edit

(c) Select the Use Cases to Include in your Product

Cardinality	Variant Use Case	Type
min=1, max=1	☑ UC2	Mandatory
min=1, max=2	☑ UC3	Optional
	□ UC4	Optional

(d) Select which Variation Point to Process

Including Use Case	Variation Point	Type
UC1	● VP1	Mandatory
	○ VP4	Optional
UC2	○ VP2	Mandatory
UC3	○ VP3	Mandatory

Figure 4.17. PUMConf's User Interface for (a) selecting variant use cases for the corresponding variation point, (b) explaining the contradicting decisions, (c) updating the contradicting decision, and (d) showing the updated list of variation points after the updated decision.

4.9.1.2 Study Context

IEE is a typical supplier in the automotive domain, producing sensing systems (e.g., Vehicle Occupant Classification, Smart Trunk Opener, and Driver Presence Detection) for multiple automotive manufacturers. In IEE's business context, like in many others, use cases are central development artifacts which are used for communicating requirements among stakeholders, such as customers. In other words, in the context of product lines, IEE's software development practices are strongly use case-driven and analysts elicit requirements and produce a new version of use cases for each new customer and product. As a result, IEE needs to adopt PLE concepts (e.g., variation points and variants) to identify commonalities and variabilities early in requirements analysis. These concepts are essential for communicating variability to customers, documenting it for software engineers, and supporting decision making during the elicitation of customer specific requirements [Halmans and Pohl, 2003].

Like in many other environments, the current practice at IEE is based on *clone-and-own* reuse [Clements and Northrop, 2001]. IEE starts a new product family with an initial customer providing requirements of a single product in the product family. The initial product requirements are elicited and documented as use cases and a domain model which are copied and then maintained for each new customer. Changes are then made manually in the copied models.

IEE provided their initial STO documentation, which contained a use case diagram, use case specifications, and supplementary requirements specifications describing non-functional requirements and domain concepts. The initial documentation was the output of their current clone-and-own reuse practice. That documentation contains variability information only in the form of some brief textual notes attached to the relevant use case specifications.

To model the STO requirements according to our product line use case modeling method, PUM, we first examined the initial STO documentation. Since the initial documentation contains almost no structured variability information, we had to work together with IEE engineers to build and iteratively refine our models (see Chapter 3). When we started to study the STO documentation, the STO project was in its initial phase and there was only one prototype implementation to discuss with some potential customers. One may argue that it is not always easy to identify variations in requirements when a new project starts. However, the IEE analysts stated that, most of the time in their domain of applications, requirements and their variability can be identified with the first customer.

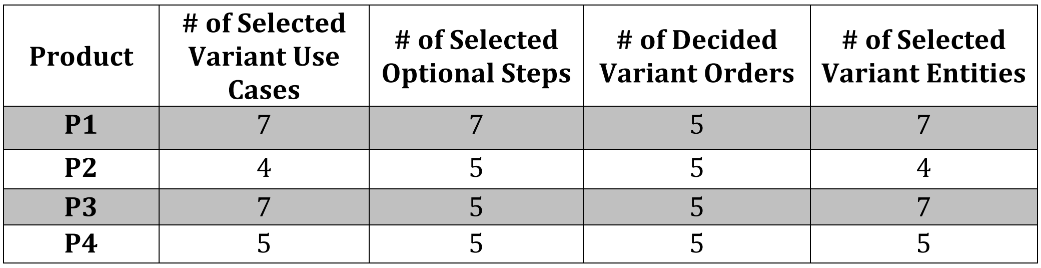
4.9.1.3 Results

After studying the initial STO documentation and meeting with the IEE analysts, we built the PL use cases and domain model for STO. The diagram in Fig. 3.5, the use case specifications in Table 3.2, and the domain model in Fig. 3.6 are part of the PL models we derived as a result of our modeling effort. Tables 3.4 and 3.5 report on the size of the entire PL use cases and domain model for STO.

As we discussed in Chapter 3 in detail, our modeling method, as part of our configuration approach, provided better assistance for capturing and analyzing variability information compared to the current, more informal practice at IEE. With the PL extensions, for example, we could unveil variability information not covered in the initial STO documentation. For instance, the use case diagram extensions helped us identify and model that *Clear Error Status via IEE QC Mode* is mandatory while *Clear Error Status via Diagnostic Mode* is optional (see Fig. 3.5), which was not previously documented.

When discussions start with a customer regarding a specific product, the IEE analysts need to make decisions on variability aspects documented in PL use case models. At a later stage, when we met again with the IEE analysts for discussing configuration needs, IEE had already developed various STO products for different car manufacturers. By using PUMConf, we, together with the IEE analysts, configured the PS use case and domain models for four products selected among the STO products IEE had already developed. The IEE analysts made the configuration decisions on the PL models using the guidance provided by PUMConf. Table 4.3 summarizes the results of the configuration for the STO products using our approach.

Table 4.3. Results Summary for the Configuration of STO Use Case and Domain Models for Various Car Manufacturers



Product	# of Selected Variant Use Cases	# of Selected Optional Steps	# of Decided Variant Orders	# of Selected Variant Entities
P1	7	7	5	7
P2	4	5	5	4
P3	7	5	5	7
P4	5	5	5	5

The first column is the number of variant use cases selected by the analysts for each product. In the PL use case diagram, there are six variant use cases with a mandatory variability relation, which are automatically selected. Table 4.3 does not include the automatically selected variant use cases and the essential use cases. The PL specifications have twenty-seven optional steps. Among them, five optional steps have a variant order to be decided (see the variant use case *Provide System User Data via Standard Mode* in Table 3.2). The fourth column in Table 4.3 presents the number of entities selected among twelve variant entities (see Table 3.5).

All the generated PS use case and domain models were confirmed by the IEE analysts to be correct and complete. The PL models that we derived from the initial STO documentation were sufficient to make all the configuration decisions needed in PUMConf to generate the correct and complete PS models for the STO products.

4.9.2 Questionnaire Study

We conducted a questionnaire study to evaluate, based on the viewpoints of IEE engineers, how well our configuration approach addresses the challenges that we identified in capturing requirements variability and configuring PS use cases. The study is described and reported according to the template provided by Wohlin et al. [Wohlin et al., 2012].

4.9.2.1 Planning and Design

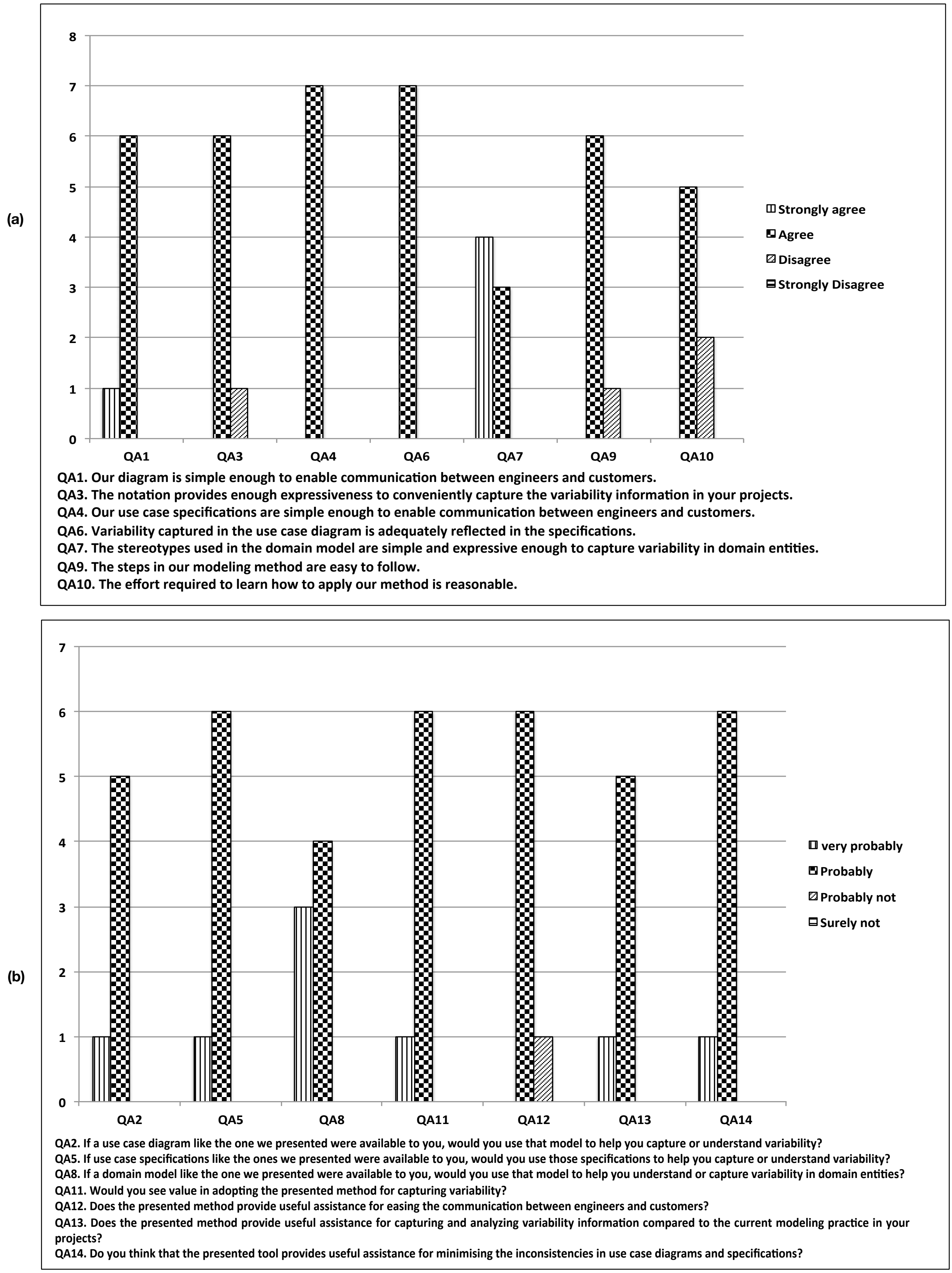
To evaluate the output of PUMConf in light of the challenges we identified earlier, we had a semi-structured interview with seven participants holding various roles at IEE: software development manager, software team group leader, software lead engineer, system engineer, and embedded software engineer. All participants had experience with use case-driven development and modeling. The interview was preceded by presentations illustrating the PL extensions of use case and domain models, PUMConf steps, a tool demo, and detailed examples from STO. Interactive sessions included questions posed to the participants about the models. In the sessions, the participants took a more active role and gave us feedback. We also organized three hands-on sessions in which the participants could apply the proposed modeling method and the PUMConf tool. In the first hands-on session, the participants were asked to find inconsistencies in the faulty PL use case diagram and specifications. In the second session, they were using PUMConf to identify and resolve contradicting configuration decisions in the STO PL use case and domain models. In the third session, the participants used PUMConf to configure PS use case and domain models from the STO PL models.

To capture the perception of engineers participating in the interviews, regarding the potential benefits of PUMConf and how it addresses the targeted challenges, we handed out two questionnaires including questions to be answered according to two Likert scales [Oppenheim, 2005] (i.e., agreement and probability). The questionnaires were structured for the participants to assess our modeling method and our configurator, PUMConf, in terms of adoption effort, expressiveness, comparison with current practice, and tool support. The participants were also encouraged to provide open, written comments.

4.9.2.2 Results and Analysis

We solicited the opinions of the participants using questionnaires (see Fig. 4.18 and Fig. 4.19). The objective of the first questionnaire was to assess our product line use case modeling method. Fig. 4.18(a) and (b) depict the questions and answers from the participants for the first questionnaire.

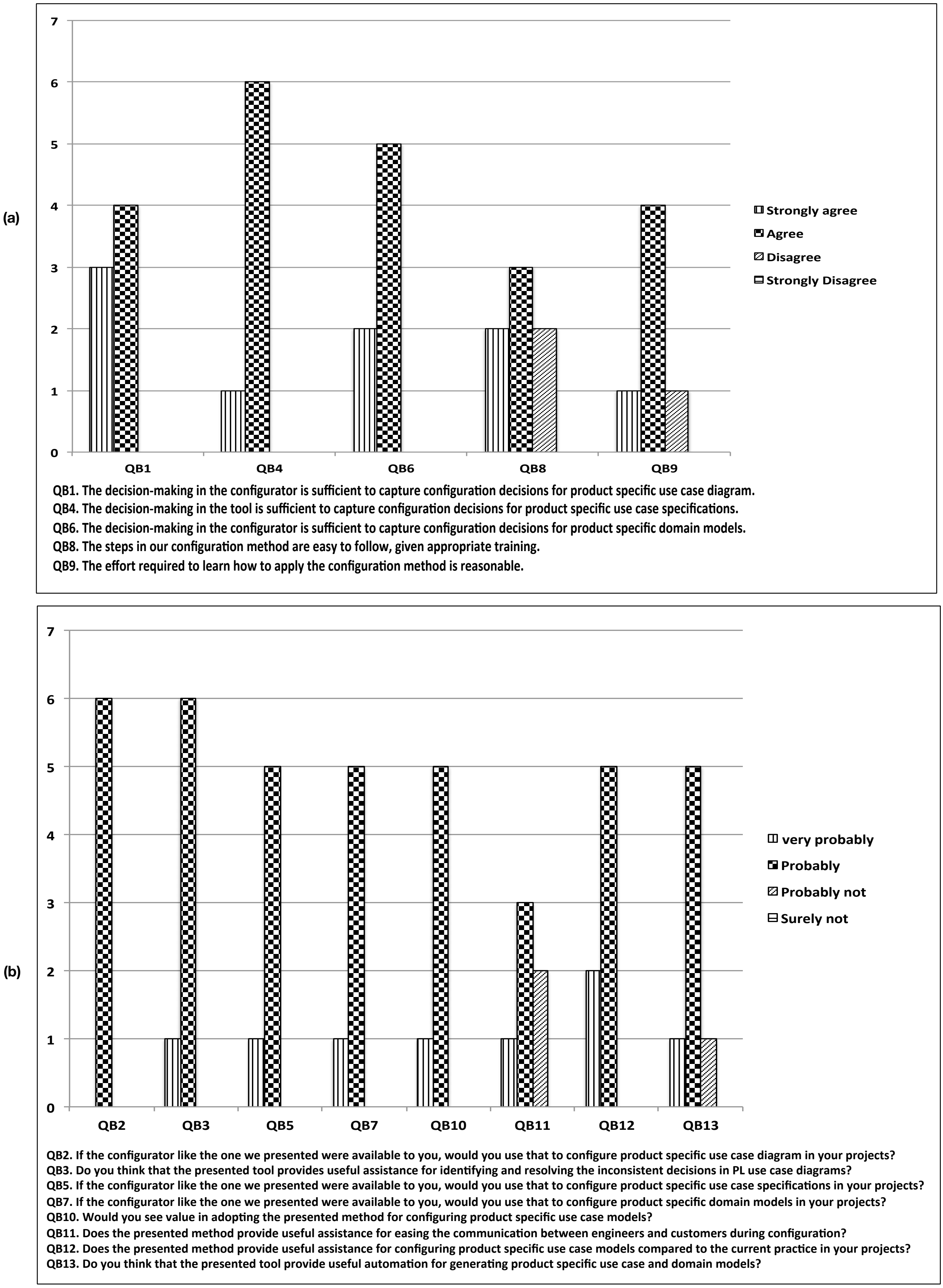
All participants agreed that the PL extensions for use case diagrams are simple enough to enable communication between engineers and customers (*QA*1) and thus they would probably use such extensions in their projects (*QA*2). Except for one case, all participants agreed that the extensions provide enough expressiveness to capture variability information in their projects (*QA*3). The participant who disagreed on *QA*3 commented that a few customers do not employ use cases in their development practice and, in these cases, the IEE analysts opt for informal discussions about the product. However, in all cases, use cases are nevertheless employed at IEE as part of their internal practice. All participants stated that the PL specification extensions are simple enough (*QA*4) and variability captured in the diagram is adequately reflected in the specifications (*QA*6). They also all agreed that they would use the specification extensions to capture variability information (*QA*5). There was also a strong consensus among the participants about expressiveness and simplicity of the PL domain model extensions (*QA*7 and *QA*8). The last part of the questionnaire focuses on the overall modeling method in terms of expressiveness, usefulness and adoption effort (*QA*9 - *QA*14). The participants provided a very positive feedback for the method in general but they also stated that (i) additional practice and training was still needed to become familiar with the method and tool support, (ii) cus-



QA1. Our diagram is simple enough to enable communication between engineers and customers.
QA3. The notation provides enough expressiveness to conveniently capture the variability information in your projects.
QA4. Our use case specifications are simple enough to enable communication between engineers and customers.
QA6. Variability captured in the use case diagram is adequately reflected in the specifications.
QA7. The stereotypes used in the domain model are simple and expressive enough to capture variability in domain entities.
QA9. The steps in our modeling method are easy to follow.
QA10. The effort required to learn how to apply our method is reasonable.

QA2. If a use case diagram like the one we presented were available to you, would you use that model to help you capture or understand variability?
QA5. If use case specifications like the ones we presented were available to you, would you use those specifications to help you capture or understand variability?
QA8. If a domain model like the one we presented were available to you, would you use that model to help you understand or capture variability in domain entities?
QA11. Would you see value in adopting the presented method for capturing variability?
QA12. Does the presented method provide useful assistance for easing the communication between engineers and customers?
QA13. Does the presented method provide useful assistance for capturing and analyzing variability information compared to the current modeling practice in your projects?
QA14. Do you think that the presented tool provides useful assistance for minimising the inconsistencies in use case diagrams and specifications?

Figure 4.18. Responses to the Questions Related to the Product Line Use Case Modeling Method



QB1. The decision-making in the configurator is sufficient to capture configuration decisions for product specific use case diagram.
QB4. The decision-making in the tool is sufficient to capture configuration decisions for product specific use case specifications.
QB6. The decision-making in the configurator is sufficient to capture configuration decisions for product specific domain models.
QB8. The steps in our configuration method are easy to follow, given appropriate training.
QB9. The effort required to learn how to apply the configuration method is reasonable.

QB2. If the configurator like the one we presented were available to you, would you use that to configure product specific use case diagram in your projects?
QB3. Do you think that the presented tool provides useful assistance for identifying and resolving the inconsistent decisions in PL use case diagrams?
QB5. If the configurator like the one we presented were available to you, would you use that to configure product specific use case specifications in your projects?
QB7. If the configurator like the one we presented were available to you, would you use that to configure product specific domain models in your projects?
QB10. Would you see value in adopting the presented method for configuring product specific use case models?
QB11. Does the presented method provide useful assistance for easing the communication between engineers and customers during configuration?
QB12. Does the presented method provide useful assistance for configuring product specific use case models compared to the current practice in your projects?
QB13. Do you think that the presented tool provide useful automation for generating product specific use case and domain models?

Figure 4.19. Responses to the Questions Related to the Configuration Approach

tomers also needed to be trained regarding the extensions, RUCM and the tool support, and (iii) the software development of a few customers is not use case-driven. These were the reasons stated for the disagreement of two participants on *QA*10 and *QA*12.

The objective of the second questionnaire was to assess our use case-driven configuration approach and its tool support. Fig. 4.19(a) and (b) depict the corresponding answers. The second questionnaire was structured in four different parts: configuring the PS use case diagram (*QB*1 - *QB*3), configuring the PS use case specifications (*QB*4 and *QB*5), configuring the PS domain model (*QB*6 and *QB*7), and the overall configuration approach and tool support (*QB*8 - *QB*13). All participants agreed that the configurator was adequate to capture configuration decisions for PS use cases and domain models, and they would use the configurator to configure PS models in their projects (*QB*1 - *QB*7). For the overall configuration approach and tool support (*QB*8 - *QB*13), two participants raised issues similar to those of the first questionnaire. Customers also need training to get familiar with the configuration approach and tool support (*QB*8 and *QB*9). Since a few customers do not rely on use case modeling, IEE analysts would in these cases use the configurator only for internal communication and documentation during product development (*QB*11). On the other hand, all participants saw value in adopting the configuration approach (*QB*10), and they agreed that the configurator provides useful assistance for configuring PS use case and domain models, compared to the current practice in their projects (*QB*12).

4.9.2.3 Threats to Validity

The main threat to validity in our case study concerns the generalizability of conclusions. To mitigate this threat, we applied PUMConf to an industrial case study that includes nontrivial use cases in an application domain with multiple customers and numerous sources of variability. Though their number is small, we selected the respondents to our questionnaire and interviews to hold various, representative roles and with substantial industry experience. We can also confidently say that the software development practice at IEE is typical of embedded system development in the automotive domain. To limit threats to internal validity, we had many meetings with the IEE analysts in the STO project to verify the correctness and completeness of our models.

4.9.3 Discussions with the Analysts and Engineers

The questionnaire study had open, written comments under each section, in which the participants could state their opinions in a few sentences about how PUMConf addresses the challenges reported in Section 4.2. As reported in Section 4.9.2, the participants' answers to the questions through Likert scales and their open comments indicate that they see high value in adopting the configuration approach and its tool support in an industrial setting in terms of increasing reusability, minimizing modeling effort, and providing effective automation. In order to elaborate over the open comments in the two questionnaires, we organized further discussions with the participants. Based on the feedback in the comments, we identified three aspects to discuss with the participants: *modeling effort*, *degree of automation*, and *limitations of the configuration approach*.

4.9.3.1 Modeling Effort

In the current practice at IEE, like in many other environments, there is no systematic way to model variability information in use case and domain models. As mentioned before, the IEE analysts take only brief notes attached to use case specifications to indicate what may vary in the specification. They are reluctant to use feature models traced to use case specifications because of two main issues: (i) having feature models requires considerable additional modeling effort with manual assignment of traces at a very low level of granularity, e.g., sequences of use case steps; and (ii) they find it hard and distracting to switch from feature models to use cases and vice versa during the decision-making process. The PL extensions in Chapter 3 enable the analysts to model variability information directly in use case and domain models without any feature modeling. The IEE analysts stated that the effort required to apply the extensions for modeling variability information was reasonable. By having variability information in use case and domain models, the analysts could focus on one artifact at a time to make configuration decisions. They considered the extensions to be simple enough to enable communication between analysts and customers, but they also mentioned that training customers may be more of a challenge since the company may need customers' consent to adopt PUMConf. Thus, its costs and benefits should be made clear to customers.

4.9.3.2 Degree of Automation

In our discussions with the analysts at IEE, we noticed that: (i) the current clone-and-own reuse practice has no systematic way and automated support to decide what to include in PS use case and domain models; (ii) typically, multiple analysts and engineers from both the customer and supplier sides are involved in the decision-making process; (iii) the analysts and engineers have to spend several days to manually review the entire set of requirements cloned from the previous product; and (iv) the intended updates on the cloned use case and domain models are manually carried out by the IEE analysts. On the other hand, PUMConf consists of various automated use case modeling and configuration activities in the context of product lines. The decision making process is automated in the sense that the IEE analysts are guided through the PL artifacts for collecting and verifying configuration decisions, while the generation of PS artifacts does not require any human intervention. Using PUMConf, the IEE analysts only select a set of relevant variant use cases, optional steps, and variant entities for a product. Corresponding PS use case and domain models are obtained from the PL models automatically, which greatly reduces the complexity of the entire configuration process. Though modeling variability in PL models is mostly manual, PUMConf provides automatic consistency checking for these models and feedback to the analyst to help them refine and correct the models. The IEE analysts considered the automated consistency checking of decisions and the generation of PS artifacts to be highly valuable.

4.9.3.3 Limitations of the Configuration Approach

Our configuration approach and tool support have some limitations at this current stage. First, our modeling method supports only functional requirements. As stated by IEE analysts, there are numerous types of non-functional requirements (e.g., security, timing, and reliability) which may play a key role in variability associated with functional requirements. It is crucial to capture and configure such aspects as well. Second, we do not address and support the evolution of PL use case and domain models (see Chapter 7). When a new project starts, requirements and their variations might not be fully known. As a result, in early stages, analysts are expected to redefine variation points and variants in requirements specifications through frequent iterations. We were told such changes need to be managed and supported to enable analysts to converge towards consistent and complete requirements and variability information. Third, PUMConf is currently implemented as a plugin in IBM DOORS, in combination with commercial modeling tools used at IEE, i.e., IBM Rhapsody, and Papyrus. PUMConf highly depends on the outputs of these tools. The analysts mentioned that these tools might be replaced with other tools or the newer versions of the same tools in the future. Future changes in the tool chain will need to fulfill the following constraints: (i) a new modeling tool for PL diagrams should be extensible in such a way that we can implement the PL diagram extensions in its use case metamodel, (ii) a new requirements management tool should not enforce its own template and restriction rules that conflict with the RUCM and PL specification extensions, and (iii) a new tool for domain modeling should support the profiling mechanism which enables analysts to model the domain with the PL stereotypes.

4.10 Conclusion

This chapter presented a configuration approach that is dedicated to environments relying on use case-driven development. It guides customers in making configuration decisions and automatically generates use case diagrams, use case specifications, and domain models for configured products. Our main motivations are to provide a high degree of automation during configuration and to rely exclusively on variability modeling for commonly used artifacts in use case-driven development, thus avoiding unnecessary modeling overhead and complexity. Our configuration approach builds on our previous work (i.e., Product line Use case Modeling method) and is supported by a tool relying on natural language processing and integrated into IBM DOORS, that aims at (1) checking artifact consistency, (2) identifying partial order of decisions to be made, (3) detecting contradicting decisions, and (4) generating product-specific use case and domain models. The key characteristic of our approach is that variability is directly captured in product line use case and domain models, at a level of granularity enabling both precise communication with various stakeholders, at different levels of details, and automated product configuration. We performed a case study in the context of automotive embedded system development. The results from structured interviews and a questionnaire study with experienced engineers suggest that our approach is practical and beneficial to configure product use case and domain models in industrial settings.

In our current tool support, we assume that product-line use case diagrams are configurable, i.e., there is at least one valid product use case diagram that can be generated from a product-line diagram. We further plan to improve the proposed approach and configurator (PUMConf) to identify non-configurable product-line diagrams before making configuration decisions.

Apart from the product line extensions, which are independent from any application domain, our approach does not make use of any further extensions specific to the embedded, automotive domain. Therefore, PUMConf should be applicable to other domains and should not require any significant adaptation as long as software development is use case-driven.

PUMConf does not currently support the detection of decisions violating constraints imposed by the variant dependencies in PL domain models. We plan to improve the tool for detecting such decisions, which is very similar to the detection of contradicting decisions in PL use case diagrams.

For resolving contradicting decisions in PL use case diagrams, our approach follows the strategy in which the detected contradiction is fixed right away and the configuration is returned to a consistent state. Our motivation stemmed from the observation that, in the considered business context, tolerating contradictions in decision-making significantly increases the complexity of communication with customers. On the other hand, from a general standpoint, fixing a contradiction immediately may not always be the optimal solution. Therefore, we plan to extend our approach to support multiple contradiction resolution strategies.

We assume that complete product specific models are generated from product line models in a single (albeit complex and iterative) stage, which is performed jointly by a team that combines the activities of all the different parties involved in the configuration process. This is a valid assumption in the observed business context, as well as in many other similar contexts. Given this assumption, our tool provides the possibility of backtracking to change configuration decisions made in preceding steps of the same stage. However, other organizations may require a multi-stage configuration process, involving multiple physically dispersed configuration teams, which have to perform their configuration steps in a pre-defined order. Each step may need to be performed by different experts at different times in physically different locations, possibly using different configuration tools. In such multi-stage configuration, supporting backtracking to earlier stages for resolving contradictions may not be an option.

Our evaluation does not address the usability of PUMConf, especially in terms of resolving contradicting decisions. As future work, we plan to conduct an extensive user study with engineers to evaluate the effort that needs to be made in resolving contradicting decisions in PUMConf.

In this chapter, we answered *Research Question 3*: *To what extent and how can we automate the interactive configuration of use case and domain models? How can we support the analysts for making configuration decisions and for generating PS use case and domain models?* Our configuration algorithm, supporting the identification of decisions contradicting prior decisions, supports interactive decision making and generation of PS use case and domain models.

PUMConf is only a first step to achieve our long term objective, i.e., change impact analysis and regression test selection in the context of use case-driven development and testing. Change can occur both in configuration decisions and variability aspects of product-line models. For decision changes in a product, the impact on other decisions needs to be assessed and re-configuration should be considered in the product-specific model. Further, the impact on the execution of test cases should be assessed. In contrast, changes on product-line use case models require impact assessment on decisions for each individual product and may entail reconfiguration and regression test selection in several products.

The results in this chapter are the input for change impact analysis and regression testing approaches for evolving configuration decisions. Chapter 5 presents a change impact analysis approach for evolving configuration decisions. PUMConf is extended with features in order to identify the impact of decision changes on other decisions and to incrementally reconfigure the PS use case and domain models.

Chapter 5

Change Impact Analysis for Evolving Configuration Decisions

In this chapter, we propose, apply, and assess a change impact analysis approach for evolving configuration decisions in Product Line (PL) use case models. In Chapter 3, we developed Product line Use case modeling Method (PUM) to support variability modeling in PL use case diagrams and specifications. We also developed a use case configurator, PUMConf, which interactively receives configuration decisions from users to generate Product Specific (PS) use case models from PL models (Chapter 4). Our approach is built upon PUM and PUMConf. It provides: (1) automated support to identify the impact of decision changes on prior and subsequent decisions in PL use case diagrams and (2) automated incremental regeneration of PS use case models from PL models and evolving decisions. Our tool support is an extension of PUMConf integrated with IBM Doors. Our approach has been evaluated in an industrial case study, which provides evidence that it is practical and beneficial to analyze the impact of decision changes and to incrementally regenerate PS models in industrial settings.

5.1 Introduction

Chapter 4 proposed and assessed a use case configurator, *PUMConf*, which interactively receives configuration decisions from users to generate Product Specific (PS) use case models from Product Line (PL) use case models. Requirements evolution results in changes in configuration decisions, e.g., a selected variant use case being unselected for a product. It is critical for the analysts to identify in advance the impact of such evolution for better decision-making during the configuration process. For instance, impacted decisions, i.e., subsequent decisions to be made and prior decisions cancelled or contradicting when a decision changes, need to be identified to reconfigure the generated use case models.

In this chapter, we propose, apply and assess a change impact analysis approach, based on our use case-driven modeling and configuration techniques, to support the evolution of configuration decisions. We do not address here evolving PL use case models, which need to be treated in a separate approach. Change impact analysis provides a sound basis to decide whether a change is worth the effort and which decisions should be changed as a consequence [Passos et al., 2013]. In our context, we aim to automate the identification of decisions impacted by changes in configuration decisions on PL use case models.

Our approach supports three activities. First, the analyst proposes a change but does not apply it to the corresponding configuration decision. Second, the impact of the proposed change on other configuration decisions for the PL use case diagram are automatically identified. In the PL use case diagram, variant use cases and variation points are connected to each other with some dependencies, i.e., *require*, *conflict* and *include*. In the case of a changed diagram decision contradicting prior and/or subsequent diagram decisions, such as a subsequent decision resulting in selecting variant use cases violating some dependency constraints because of the new/changed decision, we automatically detect and report them. To this end, we improved our consistency checking algorithm in Chapter 4, which enables reasoning on subsequent decisions as part of our impact analysis approach. The analyst is informed about the change impact on decisions for the PL use case diagram. Based on this, the analyst should decide whether the proposed change is to be applied to the corresponding decision. Third, the PS use case models are incrementally regenerated only for the impacted decisions after the analyst actually makes all the required changes. To do so, we implemented a model differencing pipeline which identifies decision changes to be used in the reconfiguration of PS models. There are two sets of decisions: (i) the set of previously made decisions used to initially generate the PS use case models and (ii) the set of decisions including decisions changed after the initial generation of the PS models. Our approach compares the two sets to determine for which decisions we need to incrementally regenerate the PS models. To support these three activities, we extended our configurator, PUMConf. To summarize, the contributions of this chapter are:

- A change impact analysis approach that informs the analysts about the impact of changes in configuration decisions on PL use case models in order to improve the decision-making process, and that incrementally reconfigures the generated PS use case models only for the impacted decisions;
- Publicly available tool support integrated with IBM DOORS as a plug-in, which automatically identifies the impact of configuration decision changes and incrementally regenerates the PS use case models;
- An industrial case study demonstrating the applicability and benefits of our change impact analysis approach.

In this chapter, we answer *Research Question 3 (To what extent and how can we automate the interactive configuration of use case and domain models? How can we support the analysts for making configuration decisions and for generating PS use case and domain models?)*, *Research Question 4 (What are the change scenarios for use case models and system test cases in a product family? What is necessary for these change scenarios to be handled in the configuration process? Which solutions can be used?)*, and *Research Question 5 (How can a change in a configuration decision be propagated to other decisions in PL use case models and to system test cases? How can we support the analysts in performing changes? How can we reconfigure PS use case and domain models for decision changes? How can we select and prioritize system test cases for such changes?).* With the change impact analysis approach, we address the issues about automated configuration, change scenarios for configuration decisions, change propagation in configuration decisions, and the reconfiguration of PS use case models.

This chapter is structured as follows. Section 5.2 introduces the industrial context of our case study to illustrate the practical motivations for our approach. Section 5.3 discusses the related work in light of the industrial needs identified in Section 5.2. In Section 5.4, we provide an overview of the approach. Sections 5.5 and 5.6 provide the details of its core technical parts. In Section 5.7, we present our tool while Section 5.8 reports on an industrial case study, i.e., STO, along with results and lessons learned. In Section 5.9, we conclude the chapter.

5.2 Motivation and Context

Our change impact analysis approach is developed as an extension of our configurator, PUMConf, in the context of embedded software systems interacting with multiple external systems, configured for multiple customers, and developed according a use case-driven process. In such a context, configuration decisions frequently change due to technological developments and evolving business needs. A change impact analysis approach is therefore needed for identifying other impacted decisions for the reconfiguration of PS models.

Changes on configuration decisions may have impact on other decisions in various ways. For instance, in the PL diagram in Fig. 3.5, the analyst changes the decision for the variation point *Clear Error Status* in order to resolve the contradiction with the prior decision for the variation point *Store Error Status* (see Chapter 3). This is done by selecting the variant use case *Clear Error Status*, which was previously unselected. This change has the following consequences: (i) the variation point *Method of Clearing Error Status* should now be considered in subsequent decisions; (ii) the variant use case *Clear Error Status via IEE QC Mode* is automatically selected because of the mandatory variability relation; (iii) the newly selected use cases should be added to the the PS use case diagram while the corresponding use case specifications should be added to the PS specifications; and (iv) new optional steps and alternative flows are introduced for consideration if there is any in the added specifications. In some cases, subsequent decisions are also impacted because of decision restrictions. Assume that the variant use case *Store Error Status* in the variation point *Storing Error Status* is unselected, and no decision has been made yet for the variation point *Clearing Error Status*. When the analyst changes the decision by selecting *Store Error Status*, the subsequent decision for the variation point *Clearing Error Status* is restricted because *Clear Error Status* should be selected in the subsequent decision to avoid further decision contradictions.

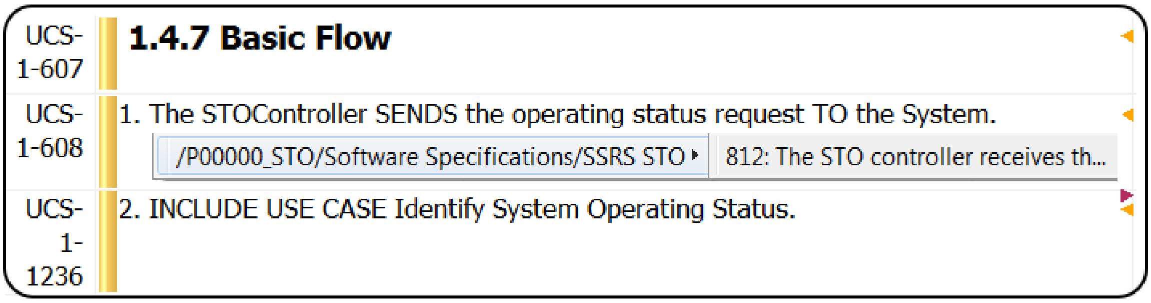
In practice, from a more general standpoint, the analysts should be aware of the impacts of decision changes to possibly reconsider some of them. After changing a decision, impact analysis support is needed to guide subsequent decisions or to change prior decisions. Within our context, we identify two challenges that need to be considered in identifying the impact of decision changes and supporting the reconfiguration of PS use case models:

***Challenge 1*: Identifying Change Impact on Prior and Subsequent Decisions for PL Use Case Diagrams.** Changes on configuration decisions for the PL use case diagram have an impact on prior decisions as well as on subsequent decisions to be made. Therefore, the impact analysis should check (i) if the change causes any contradiction with prior decisions, (ii) if there is any prior decision which becomes invalid, (iii) if any additional variation point should be considered for decision-making, and (iv) if any subsequent decision should be restricted to ensure the consistency of decisions.

***Challenge 2*: Incremental Regeneration of PS Use Case Models.** In practice, for a variety of reasons, the analysts manually assign traces from the PS use case models to other software and hardware specifications as well as to the customers' requirements documents for external systems [Ramesh and Jarke, 2001]. For instance, in order to verify the interaction between the system and the external systems, IEE's customers require that traces be assigned from the PS use case specifications to the related, external system requirements. Fig. 5.1 gives part of the basic flow of a PS use case specification in IBM DOORS with a trace to a customer's requirements specification.

Let us consider the trace in Fig. 5.1, which is from the first step of the basic flow to an external system requirement in the customer's software requirements specification. This use case step describes the operating status request sent by the STO controller, i.e., an external system implemented by the customer, while the traced external system requirement describes the condition in which the STO controller sends this request to the system. When the PS use case models are reconfigured for all the decisions, including unimpacted decisions, manually assigned traces such as the one in Fig. 5.1 are lost. The analysts need to reassign all the traces after each reconfiguration. It is therefore vital to enable the incremental regeneration of PS models by focusing only on impacted decisions. As a result, the analysts would reassign traces only for the parts of the PS use case models impacted by decision changes.

In the remainder of this chapter, we focus on how to best address these challenges in a practical



UCS-1-607	1.4.7 Basic Flow
UCS-1-608	1. The STOController SENDS the operating status request TO the System. /P00000_STO/Software Specifications/SSRS STO ▸ 812: The STO controller receives th...
UCS-1-1236	2. INCLUDE USE CASE Identify System Operating Status.

Figure 5.1. Part of an STO Use Case Specification with Trace Links

manner, in the context of use case-driven development, while relying on PUM for modeling PL use case models, and on PUMConf for the configuration of PS use case models.

5.3 Related Work

We cover the related work across four categories.

Reasoning Approaches for Product Lines. PL use case diagrams and feature models have similar modeling constructs to represent system variability in terms of variation points, variant cardinalities and dependencies. In a literature review on automated analysis of feature models [Benavides et al., 2010], three types of analysis operations on feature models are addressed: *corrective explanations*, *dependency analysis* and *valid partial configuration*. Our change impact analysis approach relies on a form of *dependency analysis* to identify the impact of changing configuration decisions in PL use case diagrams (*Challenge 1*). The *dependency analysis* operation takes a variability model (i.e., a feature model) and a partial configuration as input and returns a new configuration with the variants (i.e., features) that should be selected and/or unselected as a result of the dependency constraints [Benavides et al., 2010]. The FaMa formaL frAMEwork (FLAME) proposed by Durán et al. [Durán et al., 2016] specifies the semantics of the analysis operations, e.g., validity of a product, the set of all valid products and validity of a configuration, which can be employed not only for feature models, but also for other variability modeling languages. However, in FLAME, change impact analysis has not been considered as an analysis operation with its semantics in the presence of evolving configuration decisions. By using dependency constraints, in the context of PL use case modeling, our approach identifies variant use cases that should be selected or unselected as a result of a configuration decision change.

Trinidad et al. [Trinidad et al., 2008] and White et al. [White et al., 2010] [White et al., 2008] provide techniques to automatically propose decision changes when a dependency constraint is violated by some configuration decisions in a partial configuration. In contrast, our approach identifies (potential) violations of dependency constraints when the analyst proposes a configuration decision change. We can classify the automated support for the analysis operations according to the logic paradigm it relies on: *propositional logic* [Mannion, 2002] [Mannion and Camara, 2003] [Batory, 2005], *constraint programming* [Benavides et al., 2005b] [Benavides et al., 2005a] [Karatas et al., 2010] and *description logic* [Wang et al., 2005] [Wang et al., 2007] [Fan and Zhang, 2006]. Regarding propositional, a variability model is first mapped into a propositional formula in conjunctive normal form (CNF). A SAT solver takes the derived propositional formula and assumptions (configuration decisions) as input and determines if the formula is either satisfiable (SAT) or unsatisfiable (UNSAT). Techniques such as HUMUS (High-level Union of Minimal Unsatisfiable Sets) [Nöhrer et al., 2012] [Nöhrer and Egyed, 2013] are used to identify the contradicting configuration decisions in the presence of UNSAT. Although we map the PL use case diagram into propositional logic formulas, we do not employ any SAT solving technique. Instead, for reasons explained below, we develop our own impact analysis algorithm in our use case-driven product line context (see Section 5.5). When a change is introduced to a diagram decision, our algorithm checks the consistency of decisions to identify the impact on prior and subsequent decisions. A decision change can violate dependency constraints with prior decisions or restrict subsequent decisions. One important point is that our algorithm identifies not only the impacted decisions but also the reason of the impact, e.g., violation of dependency constraints, changing decision restrictions, and contradicting decision restrictions. In practice, the reason of the impact is important for the analysts to decide whether the proposed change is worth the effort and what further changes to make on impacted decisions. In contrast, when using SAT solvers, we only obtain as output, without any further explanation, decisions contradicting each other after the decision change [Nöhrer et al., 2012] [White et al., 2010]. For instance, assume that the analyst unselects the selected variant use case *Store Error Status* while there is no decision made yet for the variation point *Clearing Error Status* in Fig. 3.5. Our approach identifies that the subsequent decision for *Clearing Error Status* is impacted because the decision restriction previously introduced through the *require* dependency becomes invalid after the change.

Impact Analysis Approaches for Product Lines. In the context of product line engineering, most of the approaches in the literature focus on the evolution of variability models instead of the evolution of configuration decisions [Botterweck and Pleuss, 2014]. They predict the potential further changes in a PL model, e.g., a feature model, when deciding about a change in the same model. For instance, Thüm et al. [Thüm et al., 2009] present an algorithm to reason about feature model changes. The evolution of a feature model is classified as *refactoring* (i.e., no new products are added), *specialization* (i.e., no new products are added and some existing products removed), *generalization* (i.e., new products are added and no existing products removed), and *arbitrary edits*. The presented algorithm takes two versions of the same feature model as input and automatically computes the change classification. Alves et al. [Alves et al., 2006] provide a catalog of change operations (e.g., *add new alternative feature* and *replace mandatory feature*) for refactoring feature models. Paskevicius et al. [Paskevicius et al., 2012] employ a similar catalog of change operations to propagate a feature model change to other feature model elements through feature dependencies such as *parent* and *child*. Because the approach proposed by Thüm et al. [Thüm et al., 2009] does not identify the change operations applied between two versions, Acher et al. [Acher et al., 2012] build on it to identify the differences between feature models in terms of propositional formulas. It does so by comparing configuration spaces of the feature models. Bürdek et al. [Bürdek et al., 2015] propose a model differencing approach, which is similar to our model differencing pipeline in Section 5.6, to determine and document complex change operations between the feature model versions (i.e., feature models before and after changes). Our model differencing pipeline identifies configuration decision changes, while their approach is used to determine changes between two feature models, not between two configurations.

Seidl et al. [Seidl et al., 2012] assume that there are mappings, provided by the analyst, from feature models to artifacts such as UML class diagrams and source code. They propose a classification of feature model changes that captures the impact of these changes on the feature model mappings and the mapped artifacts. Quinton et al. [Quinton et al., 2015] propose yet another approach to ensure consistency of feature models and their mapped artifacts when feature models evolve. Dintzer et al. [Dintzner et al., 2014] compute the impact of a feature model change on the existing configurations of a product line by using partial dependency information in feature models. Similar to Dintzer et al. [Dintzner et al., 2014], Heider et al. [Heider et al., 2012b] [Heider et al., 2012a] propose another approach using regression testing to identify the impact of variability model changes on products. For a change in a variability model of a product line, the approach identifies whether configuration decisions for the existing products need to be changed as well. Then, it reconfigures all the products in the product line for the impacted decisions. The approach also compares the reconfigured products with the previous version to inform the analysts about the changed parts of the products.

One of the main differences between our approach and all the other approaches given above is that the latter mainly focus on changes on feature models, not changes on configuration decisions, while our approach deals with configuration decision changes and their impact on other decisions in PL use case models (*Challenge 1*). We incrementally reconfigure PS use case models as a result of evolving configuration decisions (*Challenge 2*) and do not address evolving PL models. White et al. [White et al., 2014] propose an automated approach for deriving a set of configurations on a feature model that meet a series of requirements in a multi-step configuration process. It is assumed that an initial configuration evolves to a desired configuration where the analysts do not know the intermediate configuration steps which involve configuration decision changes requiring multiple steps. The approach derives potential configuration paths between the initial and desired configurations by mapping them to a Constraint Satisfaction Problem (CSP). In contrast, our approach does not calculate potential configuration paths for the desired configuration but guides the analyst in addressing the impact of decision changes and ensure that a legal configuration is reached.

Another main difference is that our working context is specific to use case models with a specific product line modeling method, i.e., *PUM*, which explicitly models variability information in use case models, without any additional artifact such as feature models. The benefits of use case-driven configuration have been acknowledged and there are approaches proposed in the literature [Alves et al., 2010] [Alférez et al., 2014] [Rabiser et al., 2010]. However, to the best of our knowledge, there is no work addressing the impact analysis of evolving configuration decisions in the context of use case-driven configuration. Many configuration approaches [Alférez et al., 2009] [Zschaler et al., 2009] [Czarnecki and Antkiewicz, 2005] [Eriksson et al., 2009] [Eriksson et al., 2004] require that feature models be traced as an orthogonal model to artifacts such as UML use case, activity and class diagrams. Alternatively, we could have developed our impact analysis approach using feature models traced to use case models. In such a case, feature modeling needs to be introduced into practice, including establishing and maintaining traces between feature models and use case specifications and diagrams, as well as other artifacts. At IEE and in many other development environments, such additional modeling notations and the associated traceability are often perceived as an unnecessary overhead (see Chapter 3).

Impact Analysis Approaches for Requirements Models. There are impact analysis approaches that address change propagation in requirements, but not specifically in a product line context. Goknil et al. [Goknil et al., 2014] [Goknil et al., 2008a] [ten Hove et al., 2009] propose a change impact analysis approach which propagates changes in natural language requirements to other requirements by using the formal semantics of requirements relations, e.g., 'requires', 'refines' and 'conflicts' [Goknil et al., 2011] [Goknil et al., 2008b]. When requirements are expressed in models such as goal models, more specialized dependency types can be used for impact analysis. For instance, Cleland-Huang et al. [Cleland-Huang et al., 2005] use soft goal dependencies to analyze how changes in functional requirements impact non-functional requirements, while Amyot [Amyot, 2003] uses operationalization dependencies between use cases and goals to propagate change between intentional and behavioral requirements. Arora et al. [Arora et al., 2015a] [Arora et al., 2015b] propose another approach for impact analysis over Natural Language (NL) requirements by employing Natural Language Processing (NLP) techniques including the use of syntactic and semantic similarity measures. The approach uses similarity measures to compute the change impact in terms of relatedness between the changed requirement and other requirements in the requirements document. Nejati et al. [Nejati et al., 2016] extend the approach to propagate requirements changes to design models in SysML. Our work was inspired from the above techniques in terms of using requirements relations to propagate changes among diagram decisions (*Challenge 1*). Our approach does not address changes in natural language requirements, but deals with propagating decision changes to other decisions through variation point-variant use case dependencies in the context of use case-driven configuration.

Incremental Model Generation Approaches. Use case-driven configuration approaches in the literature (e.g., [Eriksson et al., 2005a] [Fantechi et al., 2004b] [Czarnecki and Antkiewicz, 2005] [Alférez et al., 2009]) do not support incremental reconfiguration of use cases for evolving configuration decisions (*Challenge 2*). There are also more general configuration approaches (e.g., [Dhungana et al., 2011] [Rosa et al., 2009]) that can be customized to configure PS use case models. For instance, DOPLER [Dhungana et al., 2011] supports capturing variability information as a variability model, and modeling any type of artifact as asset models. Variability and asset models are linked by using trace relations. The approach proposed by Heider et al. [Heider et al., 2012a] [Heider et al., 2012b] is an extension of DOPLER to identify the impact of changes of variability information on products. It reconfigures all the products in the product line for the impacted decisions. However, it focuses on changes in variability information, not changes in configuration decisions. It is also not incremental, limiting its applicability, as the reconfiguration encompasses all the decisions, not only the affected ones.

Considerable attention in the model-driven engineering research community has been given to incremental model generation/transformation for model changes (e.g., [Hearnden et al., 2006] [Kurtev et al., 2007] [Jahann and Egyed, 2004] [Giese and Wagner, 2009] [Xiong et al., 2007]), and this line of work has inspired initiatives in many software engineering domains. For instance, Vogel et al. [Vogel et al., 2009] use incremental model transformation techniques for synchronizing runtime models by integrating a general-purpose model transformation engine into their runtime modeling environment. Bidirectional model transformations are employed by Eramo et al. [Eramo et al., 2012] to support the synchronization and interoperability of architecture models for architecture model changes. Alternatively, we could also have employed a generic model transformation engine and language to implement the incremental generation of PS use case models. Compared to model transformation languages, in terms of loading, matching and editing text in natural language, Java provides much more flexibility for handling plain text use case specifications. As a result, we used Java to implement the generation of PS use case models in Chapter 4 but also for implementing the incremental reconfiguration of PS models as a model differencing and reconfiguration pipeline (see Section 5.6). To the best of our knowledge, our approach is the first to support incremental reconfiguration of PS use case models for evolving decisions in a product family.

5.4 Overview of the Approach

The process in Fig. 5.2 presents an overview of our approach. In Step 1, *Propose a change for a decision*, the analyst is asked to propose a change for a configuration decision made previously for the PL use case diagram.

The configuration decision change proposed by the analyst is not actually applied to the corresponding decision yet. In Step 2, *Identify the change impact on other decisions*, our approach automatically identifies the impact of the proposed change on other configuration decisions for the PL use case diagram. The analyst is informed about the impact of the decision change on prior and subsequent decisions, e.g., contradicting decisions and restricted subsequent decisions (*Challenge 1*).

The analyst evaluates the impacted decisions to decide whether the proposed change is to be applied. In Step 3, *Apply the proposed change*, the analyst applies the proposed change to the corre-

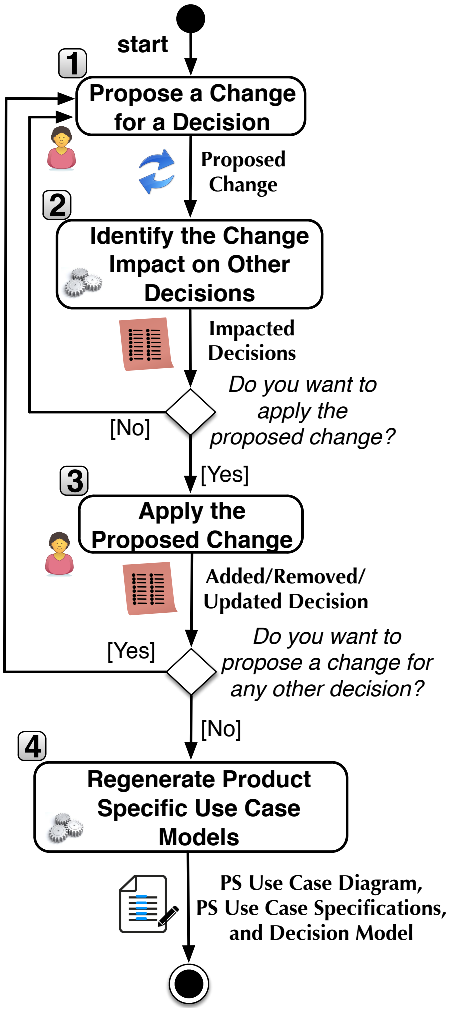

start
1
Propose a Change for a Decision
Proposed Change
2
Identify the Change Impact on Other Decisions
Impacted Decisions
Do you want to apply the proposed change?
[No]
[Yes]
3
Apply the Proposed Change
Added/Removed/ Updated Decision
Do you want to propose a change for any other decision?
[Yes]
[No]
4
Regenerate Product Specific Use Case Models
PS Use Case Diagram, PS Use Case Specifications, and Decision Model

Figure 5.2. Overview of the Approach

sponding decision. Steps 1, 2, and 3 are iterative: the analyst proposes and applies changes until all the required changes are considered. We discuss these three steps in Section 5.5.

After the analyst applies all the required changes to the configuration decisions, in Step 4, *Regenerate product specific use case models*, the PS use case diagram and specifications are incrementally and automatically regenerated for only the changed decisions (*Challenge 2*). The details of the step are described in Section 5.6.

5.5 Identification of Change Impact on Decisions for PL Use Case Diagrams

Decision-making during product configuration is iterative. The analyst may update or delete some of the prior decisions while new decisions are being made for undecided variants. A diagram decision is about selecting, for the product, variant use cases in the variation point. Table 5.1 lists the change types for diagram decisions.

Table 5.1. Change Types for Diagram Decisions

Change Types
. Add a decision
. Delete a decision
. Update a decision
- Select some unselected variant use case(s)
- Unselect some selected variant use case(s)
- Unselect some selected variant use case(s) and select some unselected variant use case(s)

The first two change types in Table 5.1 are obvious manipulations over the diagram decisions. The subtypes of 'Update a Decision' match the (un)selection of variant use cases in a variation point.

When a change is introduced to a diagram decision, the analyst needs to identify not only the impacted decisions but also the reason of the impact, e.g., violation of dependency constraints, new restrictions for subsequent decisions, and contradicting decision restrictions (*Challenge 1*).

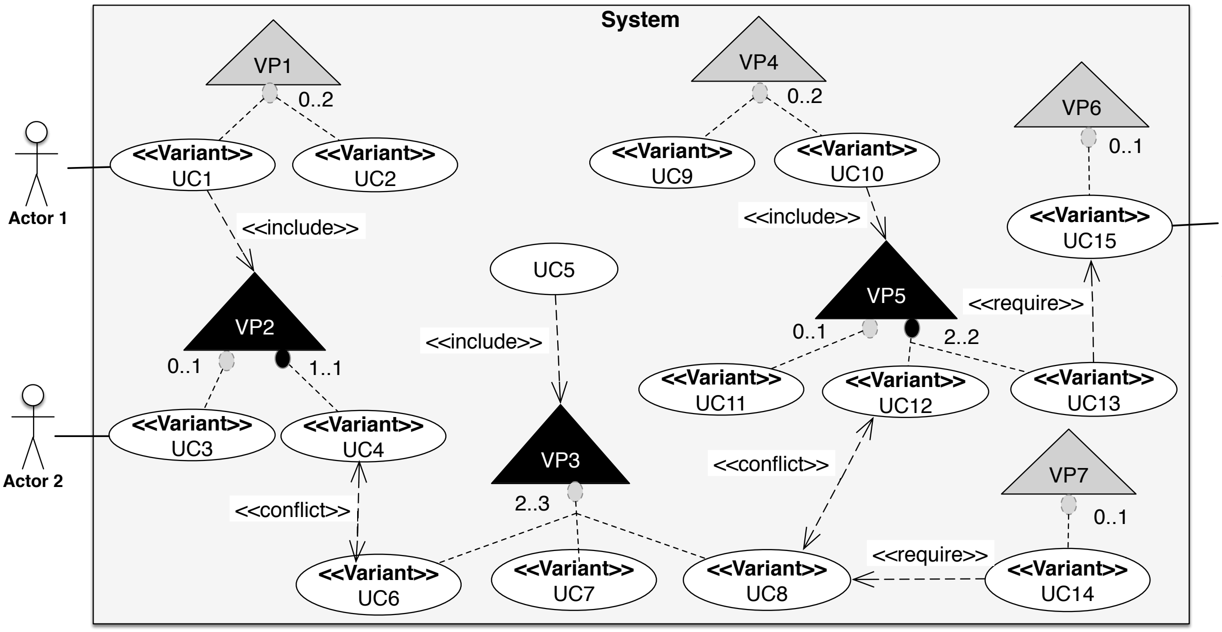


Figure 5.3. An Example Product Line Use Case Diagram

Automated analysis for configuration support often relies on translating models to propositional logic and using satisfiability (SAT) solvers [Benavides et al., 2010] [Mendonca et al., 2009b]. As we discussed in Section 5.3, employing of SAT solvers can help identify impacted decisions but does not provide further explanations regarding the reason of the impact. However, this is critical for the analysts to make further decisions based on the change impact. To this end, we devised a custom change impact analysis algorithm that identifies the impact of diagram decision changes on other diagram decisions and provide an explanation regarding the cause of the impact. In the following, we explain the steps of the algorithm with an illustrative example. The example is a slight adaptation of a piece of our industrial case study since we needed some additional modeling elements to illustrate the complete set of features of the algorithm. Fig. 5.3 depicts an example PL use case diagram including seven variation points, fourteen variant use cases, and one essential use case.

Table 5.2. Example Decisions for the PL Use Case Diagram in Fig. 5.3

Decision ID	**Explanation**
d1	Selecting UC1 and UC2 in VP1
d2	Selecting UC9 and unselecting UC10 in VP4
d3	Unselecting UC15 in VP6
d2’	Selecting UC9 and UC10 in VP4

As an example, let us assume the analyst makes the decision *d1* for *VP1*, which is selecting *UC1* and *UC2* for the product. Further, the decisions *d2* and *d3* are made for *VP4* and *VP6*, which are selecting *UC9* and unselecting *UC10* in *VP4* and unselecting *UC15* in *VP6*, respectively. Further, let us assume that the analyst proposes to change *d2* with *d2$'$* by selecting unselected *UC10* in *VP4* (see Table 5.2).

Alg. 3 describes the change impact analysis algorithm for diagram decisions. The algorithm takes a set of prior decisions, a PL use case diagram, and a decision change as input. It reports added and deleted contradicting prior decisions, added and deleted restrictions for subsequent decisions, and sets of added and deleted contradicting restrictions as output.

The decision *d*, which precedes the decision change *c*, is a quadruple of the variation point *vp*, the use case *uc* including *vp*, the set of selected variant use cases *SUC* in *vp*, and the set of unselected variant use cases *NSUC* in *vp* (Line 6). The decision *d$'$*, which results from the change *c*, is given as a similar quadruple (Line 7). For instance, in our example, *d2* and *d2$'$* are $(VP4, null, \{UC9\}, \{UC10\})$ and $(VP4, null, \{UC9, UC10\}, \emptyset)$, respectively.

We call *check* and *infer* functions with *d* and *d$'$* to identify the impact of *c* (Lines 11-16).

- ***checkPriorDecisionConsistency*** determines contradicting prior decisions for variation points. Two or more diagram decisions may contradict each other if they result in violating some variation point and variant dependency constraints (i.e., *require* and *conflict*).
- ***inferDecisionRestrictions*** determines restrictions on the selection of variant use cases in undecided variation points. The existing decisions may entail (un)selection of some variant use cases in subsequent decisions through the variation point and variant dependencies.
- ***checkDecisionRestrictions*** determines contradicting restrictions for subsequent decisions. Two or more decision restrictions may contradict each other if they result in violating some cardinality constraints or result in selecting and unselecting the same variant use case.

Alg. 3: impact

```
   Inputs : Set of prior decisions ($DC$), PL use case diagram ($PLD$), Decision change ($c$)
   Output: Sets of added and deleted contradicting prior decisions, added and deleted restrictions
           for subsequent decisions, and added and deleted sets of contradicting restrictions
1  Let $p$ be a pair $(vp, uc)$ such that $vp$ is a variation point either included by a use case $uc$, or $vp$
   is not included by any use case
2  Let $SUC$ be the set of variant use cases selected in $p.vp$ before the change $c$
3  Let $NSUC$ be the set of variant use cases unselected in $p.vp$ before the change $c$
4  Let $SUC'$ be the set of variant use cases selected in $p.vp$ after the change $c$
5  Let $NSUC'$ be the set of variant use cases unselected in $p.vp$ after the change $c$
6  Let $d$ be the quadruple $(dp.vp, dp.uc, SUC, NSUC)$
7  Let $d'$ be the quadruple $(dp.vp, dp.uc, SUC', NSUC')$
8  Let $CD$ and $CD'$ be the empty sets for contradicting decisions
9  Let $R$ and $R'$ be the empty sets for restrictions on further decisions
10 Let $CR$ and $CR'$ be the empty sets for sets of contradicting restrictions
11 $CD \leftarrow$ checkPriorDecisionConsistency($DC$, $d$, $PLD$)
12 $CD' \leftarrow$ checkPriorDecisionConsistency($DC$, $d'$, $PLD$)
13 $R \leftarrow$ inferDecisionRestrictions($DC \cup \{d\}$, $PLD$)
14 $R' \leftarrow$ inferDecisionRestrictions($DC \cup \{d'\}$, $PLD$)
15 $CR \leftarrow$ checkDecisionRestrictions($R$, $PLD$)
16 $CR' \leftarrow$ checkDecisionRestrictions($R'$, $PLD$)
17 return $(CD' \backslash CD, CD \backslash CD', R' \backslash R, R \backslash R', CR' \backslash CR, CR \backslash CR')$
```

The algorithm of *checkPriorDecisionConsistency* was developed as part of our configurator, *PUMConf*, described as part of the configuration algorithm (Alg. 1) in Chapter 4. The algorithm is based on mapping variation points, use cases and variant dependencies to propositional logic formulas. For a given decision regarding a variation point, it only checks the satisfaction of the propositional formulas derived from the dependencies of the variation point (see Chapter 4). For example, assume there are two conflicting variant use cases *Ua* and *Ub* (i.e., *Ua* conflicts with *Ub*). *Ua* and *Ub* are selected in decisions *Da* and *Db*, respectively. *Da* and *Db* are contradicting because *Ua* and *Ub* cannot exist for the same product (i.e., $\neg(Ua \wedge Ub)$).

For changing *d2* with *d2′* in Fig. 5.3, we call *checkPriorDecisionConsistency* first with *d2* ($DC = \{d1, d3\}$ and $d = d2$ in Line 11), and then with *d2′* ($DC = \{d1, d3\}$ and $d = d2'$ in Line 12). For *d2*, the function returns no contradicting prior decision. When *UC10* is unselected in *d2*, *UC11*, *UC12* and *UC13* in *VP5* are automatically unselected because there is no other use case including *VP5*. *UC13* which is unselected in *d2* requires *UC15* which is unselected in *d3* (i.e., $U13 \rightarrow U15$). Therefore, *d2* and *d3* are not contradicting. *UC12* and *UC13* are automatically selected in *d2′* because of selected *UC10* and the mandatory variability relation in *VP5*. *UC13* which is selected in *d2′* requires *UC15* which is unselected in *d3*. Therefore, for *d2′*, *checkPriorDecisionConsistency* returns *d3* contradicting *d2′*. The decision change introduces a new contradiction with *d3* ($CD' \backslash CD = \{d3\}$ in Line 17). No existing contradiction is removed by the change ($CD \backslash CD' = \emptyset$ in Line 17). *d3* is impacted since it contradicts *d2′* after the change.

As part of our impact analysis approach, the algorithm of *inferDecisionRestrictions* also relies on propositional logic mappings for variation points, use cases and variant dependencies (see Section 5.5.1 for the details of the algorithm). For a given decision regarding a variation point, *inferDecisionRestrictions* infers restrictions for subsequent decisions by only checking the satisfaction of the propositional logic formulas derived from the dependencies of the variation point. Assume two variant use cases *Ua* and *Ub* in variation points *Va* and *Vb* with a *requires* relation (i.e., *Ua* requires *Ub*). *Ua* is selected in decision *Da* for *Va* while there is no decision yet for *Vb*. The subsequent decision for *Vb* is restricted as *Ub* needs to be selected to avoid a contradiction with *Da* because of the *requires* relation (i.e., $Ua \rightarrow Ub$).

For the input decisions *d1*, *d2* and *d3*, *inferDecisionRestrictions* returns restriction *r1* for *UC6* in *VP3* (Line 13). When *UC1* is selected in *d1*, *UC4* is automatically selected because of the mandatory variability relation in *VP2*. *UC4* conflicts with *UC6*, and there is no decision made for *UC6*. The selection of *UC4* restricts the subsequent decision for *VP3* so that *UC6* should not be selected to avoid the contradiction with *d1* (i.e., *r1* in Table 5.3). For *d1*, *d2′* and *d3*, the function returns restrictions *r1* for *UC6* in *VP3*, *r2* for *UC8* in *VP3*, and *r3* for *UC14* in *VP7* (Line 14). *UC12* in *VP5* is automatically selected in *d2′*, and it conflicts with *UC8* in *VP3* for which there is no decision made yet. The selection of *UC12* restricts the subsequent decision for *VP3* through the conflicts relation that *UC8* should not be selected (i.e., *r2*). The restriction for the subsequent decision for *UC8* restricts the subsequent decision for *VP7* through the *requires* relation (i.e., *r3*). If *UC8* should not be selected, *UC14* should also not be selected since it requires *UC8*. The subsequent decisions for *VP3* and *VP7* are impacted by the change because of the new restrictions ($R' \backslash R = \{r2, r3\}$ and $R \backslash R' = \emptyset$ in Line 17).

Table 5.3. Restrictions Inferred from the Example Decisions in Table 5.2

Restriction ID	Explanation of the Restriction
r1	UC6 in VP3 should not be selected
r2	UC8 in VP3 should not be selected
r3	UC14 in VP7 should not be selected

We devise the algorithm of *checkDecisionRestrictions* as part of our change impact analysis approach (see Section 5.5.2 for the details of the algorithm). For a given set of decision restrictions, *checkDecisionRestrictions* identifies contradicting restrictions for subsequent decisions in terms of violating cardinality constraints and restricting the same variant use cases for being selected and unselected. For example, assume there are two restrictions *r1* and *r2* which state the variant use cases *Ua* and *Ub* in the variation point *V* need to be selected, respectively. *V* has the [0..1] cardinality constraint. *r1* and *r2* do not comply with this cardinality constraint.

For the restrictions before the decision change in our example (i.e., *r1*), *checkDecisionRestrictions* does not return any contradicting restriction (Line 15). *r1* restricts the subsequent decision for *VP3* so that *UC6* should not be selected. There is no other restriction, and *r1* complies with the cardinality constraint of *VP3* (i.e., [2..3]). For the restrictions after the change (i.e., *r1*, *r2* and *r3*), the function returns $\{\{r1, r2\}\}$, i.e., the set of sets of contradicting restrictions. *UC6* and *UC8* in *VP3* should not be selected according to *r1* and *r2*, respectively. The cardinality constraint in *VP3* requires at least two of three variant use cases in *VP3* to be selected. Therefore, *r1* and *r2* cannot exist together because of the cardinality constraint. A new contradiction is introduced after the decision change ($CR' \backslash CR = \{\{r1, r2\}\}$ and $CR \backslash CR' = \emptyset$ in Line 17). To resolve it, the decisions causing it need to be updated. *r2* is inferred from $d2'$ through *UC12*, while *d1* results in *r1* through *UC4*. Therefore, *d1* is identified as impacted.

Changing *d2* with $d2'$ impacts *d1* for *VP1*, *d3* for *VP6* and the subsequent decisions for *VP3* and *VP7*.

5.5.1 Identification of Subsequent Decision Restrictions

Decision restrictions are inferred by mapping variation points, use cases and variant dependencies to propositional logic formulas. We assume that a PL use case diagram *PLD* is defined as a set, where each use case is a member of the set. The PL diagram consists of *n* use cases $PLD = \{u_1, ..., u_n\}$; each use case u_i in *PLD* is represented by a boolean variable with the same name. Boolean variable u_i evaluates to *true* if use case u_i is selected and*false* otherwise. If there is no decision made yet for use case u_i, variable u_i is not valued (*unknown*). Please note that all essential use cases are automatically selected.

In Chapter 4, Fig. 4.7 provides the corresponding propositional formulas for each pattern involving dependencies, variation points, and variant use cases, where propositions capture logical relationships among variant use cases. For instance, according to the corresponding propositional formula in Fig. 4.7(a), if use case UCA_m is selected for a product then this logically implies that use case UCB_n is also selected. Fig. 4.7(c) depicts the mapping when there is a *require* dependency between two variation points *A* and *B*. In such a case, if one of the variant use cases in variation point *A* ($UCA_1 \vee ... \vee UCA_m$) is selected, then at least one of the variant use cases in variation point *B* ($\rightarrow UCB_1 \vee ... \vee UCB_n$) should also be selected.

Alg. 4 describes the algorithm for *inferDecisionRestrictions*. To illustrate the algorithm, we rely on the example with the input decisions *d1*, $d2'$ and *d3* in Fig. 5.3. For each decision *d* in the set of decisions *D*, the algorithm calls some *infer* functions to identify the decision restrictions for subsequent decisions in which the propositional logic formulas, derived from the dependencies to/from the diagram elements decided in *d*, are satisfied (Lines 11, 12, 15, 18, 19 and 21). Each *infer* function in Alg. 4 infers restrictions for subsequent decisions using the propositional formulas in one or more mappings in Fig. 4.7.

Alg. 4: inferDecisionRestrictions

```
Inputs : Set of diagram decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)

$DC \leftarrow D$
Let $IR$ be the empty set for inferred decision restrictions
while ($DC \neq \emptyset$) do
    Let $d$ be a decision in $DC$
    Let $vp$ be the variation point in decision $d$
    Let $SUC$ be the set of selected variant use cases in decision $d$
    Let $NSUC$ be the set of unselected variant use cases in decision $d$
    Let $SE$ be the set of variant use cases automatically selected when the variant use cases in
    $SUC$ are selected in decision $d$
    Let $NSE$ be the set of variant use cases automatically unselected when the variant use cases
    in $NSUC$ are unselected in decision $d$
    foreach $(u \in (SUC \cup SE))$ do
        $IR \leftarrow IR \cup$ inferRequiredByUC $(u, D, PLD)$
        $IR \leftarrow IR \cup$ inferConflictingUC $(u, D, PLD)$
    end foreach
    foreach $(u \in (NSUC \cup NSE))$ do
        $IR \leftarrow IR \cup$ inferRequiringUC $(u, D, PLD)$
    end foreach
    if ($SUC \neq \emptyset$) then
        $IR \leftarrow IR \cup$ inferRequiredByVP $(SUC, vp, D, PLD)$
        $IR \leftarrow IR \cup$ inferConflictingVP $(SUC, vp, D, PLD)$
    else
        $IR \leftarrow IR \cup$ inferRequiringVP $(NSUC, vp, D, PLD)$
    end if
    $DC \leftarrow DC \setminus \{d\}$
end while
return $IR$
```

- ***inferConflictingVP*** uses the formulas in Fig. 4.7(d) and (g) to infer decision restrictions for variation points and use cases conflicting with selected variation point *vp* in decision *d*,

- ***inferConflictingUC*** uses the formulas in Fig. 4.7(b) and (g) to infer decision restrictions for variation points and variant use cases conflicting with selected variant use case *u* in decision *d*,

- ***inferRequiringVP*** uses the formulas in Fig. 4.7(c) and (e) to infer decision restrictions for variation points and variant use cases requiring unselected variation point *vp* in decision *d*,

- ***inferRequiredByVP*** uses the formulas in Fig. 4.7(c) and (f) to infer decision restrictions for variation points and variant use cases required by selected variation point *vp* in decision *d*,

- ***inferRequiringUC*** uses the formulas in Fig. 4.7(a) and (f) to infer decision restrictions for variation points and variant use cases requiring unselected variant use case u in decision d,

- ***inferRequiredByUC*** uses the formulas in Fig. 4.7(a) and (e) to infer decision restrictions for variation points and variant use cases required by selected variant use case u in decision d.

Alg. 5: inferConflictingUC

```
Inputs : Use case ($u$), Set of decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$ and $b$ is a
boolean variable
Let $IR$ be the empty set for inferred decision restrictions
Let $CVP$ be the set of variation points conflicting with $u$
Let $CUC$ be the set of variant use cases conflicting with $u$
foreach (c ∈ CUC) do
    if ((there is a subsequent decision to be made for c) and (c has not been selected in prior decisions in D)) then
        Let $vp$ be the variation point of $c$
        $IR \leftarrow IR \cup \{(c, vp, false)\}$
        $IR \leftarrow IR \cup$ inferRequiringUC($c$, $D$, $PLD$)
        $IR \leftarrow IR \cup$ inferRequiringVP($\{c\}$, $vp$, $D$, $PLD$)
        Let $AUC$ be the set of variant use cases automatically unselected when $c$ is unselected
        foreach (a ∈ AUC) do
            Let $p$ be the variation point of $a$
            $IR \leftarrow IR \cup \{(a, p, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($a$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{a\}$, $p$, $D$, $PLD$)
        end foreach
    end if
end foreach
foreach (p ∈ CVP) do
    if ((there is a subsequent decision to be made for p) and (none of the variant use cases in p has been selected in prior
    decisions in D)) then
        Let $UC$ be the set of variant use cases in $p$
        $IR \leftarrow IR \cup \{(null, p, false)\}$
        $IR \leftarrow IR \cup$ inferRequiringVP($UC$, $p$, $D$, $PLD$)
        foreach (vc ∈ UC) do
            $IR \leftarrow IR \cup$ inferRequiringUC($vc$, $D$, $PLD$)
        end foreach
        Let $AU$ be the set of variant use cases automatically unselected when the variant use cases in $p$ are unselected
        foreach (vuc ∈ AU) do
            Let $vp$ be the variation point of $vuc$
            $IR \leftarrow IR \cup \{(vuc, vp, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($vuc$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{vuc\}$, $vp$, $D$, $PLD$)
        end foreach
    end if
end foreach
return $IR$
```

In Fig. 5.3, *inferConflictingUC* infers *r1* and *r2* from *UC4*, automatically selected in *d1*, and from *UC12*, automatically selected in *d2′*, respectively. The algorithm of *inferConflictingUC* is given in Alg. 5. For the rest of the *infer* functions, the reader is referred to Appendix B.

The algorithm of *inferConflictingUC* in Alg. 5 uses the formulas in Fig. 4.7(b) and (g) to restrict the subsequent decisions for variant use cases and variation points that conflict with selected use case *u*. For instance, in Fig. 4.7(b), when *UCAm* is selected, it checks if there is any decision made for *UCBn*. If there is no decision for *UCBn*, the subsequent decision is restricted that *UCBn* should not be selected.

inferConflictingUC takes as input selected variant use case *u*, set of decisions *D*, and PL use case diagram *PLD*, while it returns the set of decision restrictions *IR*. A decision restriction is given as a triple (uc, vpo, b) where *uc* is a variant use case, *vpo* is the variation point of *uc* and *b* is a boolean variable (Line 1 in Alg. 5). If the restriction is about the entire variation point, not about a single variant use case in the variation point, *uc* becomes *null*. *b* indicates whether the variant use case(s) should be selected or not. For instance, $(null, Va, false)$ states that none of the variant use cases in variation point *Va* should be selected, while $(UCA1, Va, true)$ states variant use case *UCA1* in *Va* should be selected.

The algorithm starts with identifying the variant use cases conflicting with the input selected variant use case *u* (see Fig. 4.7(b)). The conflicting variant use cases which have not been decided yet should be unselected in subsequent decisions (Line 8). The subsequent decisions should also be restricted for other undecided variant use cases and variation points which require those conflicting use cases (Lines 9 and 10). When the conflicting variant use cases are unselected because of the restriction, some variant use cases in the variation points included by those conflicting use cases might also be automatically unselected, and therefore the corresponding subsequent decisions need to be restricted (Lines 11-17). In our example, *UC4* is selected in *d1* (i.e., *u* = *UC4*), and only *UC6* conflicts with *UC4* (i.e., *CUC* = $\{UC6\}$ in Line 3). There is no decision made for *UC6* which should not be selected (i.e., *r1* = $(UC6, VP3, false)$ in Line 8). *UC4* does not include any variation point where variant use cases might be automatically unselected (i.e., *AUC* = $\emptyset$ in Line 12). As another input use case, *UC12* is selected in $d2'$ (i.e., *u* = *UC12*), and only *UC8* conflicts with *UC12* (i.e., *CUC* = $\{UC8\}$ in Line 3). There is no decision made for *UC8*. Therefore, it should not be selected in subsequent decisions (i.e., *r2* = $(UC8, VP3, false)$ in Line 8). *UC8* is required by *UC14* in *VP7* which has not been decided yet (see *inferRequiringUC* in Line 9). Another decision restriction *r3* is inferred for *VP7* (i.e., $(UC14, VP7, false)$).

The algorithm also identifies the variation points conflicting with the input selected variant use case *u* (see Fig. 4.7(g)). The variant use cases in the undecided conflicting variation points should be unselected in the subsequent decisions (Line 23). The variant use cases and variation points requiring those conflicting variation points or their variant use cases should also be unselected in the subsequent decisions (Lines 24-27). The subsequent decisions are restricted for variant use cases which are automatically unselected when the variant use cases in the undecided conflicting variation points are unselected (Line 28-34). For the example in Fig. 5.3, there is no variation point conflicting with the input use cases. The algorithm returns all the inferred restrictions (Line 37).

5.5.2 Identification of Contradicting Decision Restrictions

For a given set of decision restrictions, our approach identifies (i) restrictions violating cardinality constraints in variation points and (ii) contradicting restrictions regarding the selection and unselection of the same variant use case. Assume we have two restrictions *rt1* and *rt2* where *rt1* = $(null, Va, false)$ and *rt2* = $(Ua, Va, true)$. *rt1* and *rt2* contradict each other because *Ua* in *Va* should be selected according to *rt2* while *rt1* states all the variant use cases in *Va* should be unselected.

```
Alg. 6: checkDecisionRestrictions
    Inputs : Set of decision restrictions ($R$), PL use case diagram ($PLD$)
    Output: Set of sets of contradicting decisions ($CR$)
 1  Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$, and $b$ is a
    boolean variable
 2  Let $CR$ be the empty set for sets of contradicting restrictions
 3  Let $VP$ be the set of variation points in $PLD$
 4  foreach ($p \in VP$) do
 5      Let $DR$ be the set of decisions restrictions in $R$ for $p$
 6      $IR \leftarrow DR$
 7      foreach ($r \in DR$) do
 8          $IR \leftarrow IR \setminus \{r\}$
 9          foreach ($e \in IR$) do
10              if ($e.uc = r.uc$ and $e.b \neq r.b$) then
11                  $CR \leftarrow CR \cup \{\{e, r\}\}$
12              end if
13              if ($r.uc = null$) then
14                  if ($r.b = false$) then
15                      if ($e.b = true$) then
16                          $CR \leftarrow CR \cup \{\{e, r\}\}$
17                      end if
18                  end if
19              end if
20          end foreach
21          if ($r.uc = null$ and $r.b = true$ ) then
22              $CR \leftarrow CR \cup$ checkSeveralRestrictions($p$, $DR$, $PLD$)
23          end if
24      end foreach
25      $CR \leftarrow CR \cup$ checkCardinality($p$, $DR$, $PLD$)
26  end foreach
27  return $CR$
```

Alg. 6 describes the algorithm of *checkDecisionRestrictions* that identifies contradicting restrictions. A contradiction is described as a set of contradicting decisions. For each variation point *p* in the PL diagram (Lines 3 and 4), the algorithm first checks if there are multiple restrictions (Lines 10-12). A contradiction is identified for two restrictions requiring the selection and unselection of the same variant use case (Lines 11 and 16). More than two restrictions result in a contradiction where a restriction requires at least one variant use case in a variation point to be selected while each variant

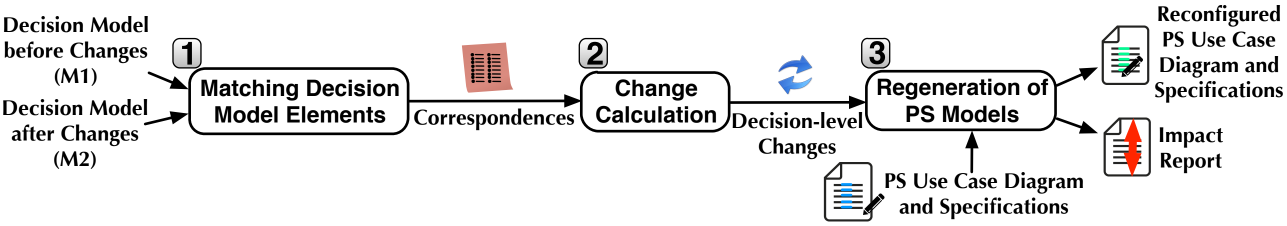

Decision Model before Changes (M1)
Decision Model after Changes (M2)
1
Matching Decision Model Elements
Correspondences
2
Change Calculation
Decision-level Changes
3
Regeneration of PS Models
PS Use Case Diagram and Specifications
Reconfigured PS Use Case Diagram and Specifications
Impact Report

Figure 5.4. Overview of the Model Differencing and Regeneration Pipeline

use case in the same variation point is required to be unselected by yet another restriction (Line 22). Restrictions which do not comply with cardinality constraints also contradict each other (Line 25). We call two functions in Alg. 6 (Lines 22 and 25).

- ***checkSeveralRestrictions*** returns a set of contradictions for restrictions in *DR* in which more than two restrictions for variation point *p* contradict each other,

- ***checkCardinality*** returns a set of contradictions for restrictions in *DR* which do not comply with the cardinality constraints in variation point *p*.

For example, *checkDecisionRestrictions* checks the example restrictions for each variation point in Fig. 5.3 where $R = \{r1, r2, r3\}$ and *PLD* is Fig. 5.3. Restrictions *r1* and *r2* apply to the subsequent decision in *VP3* while *r3* restricts another subsequent decision in *VP7*. *r1* and *r2* restrict the decision for different variant use cases in *VP3* (i.e., $r1.uc \neq r2.uc$ in Line 10, $r1.uc \neq null$ in Line 13, and $r2.uc \neq null$ in Line 13). *UC6* and *UC8* in *VP3* should be unselected according to *r1* and *r2* while the cardinality constraint requires at least two of three variant use cases in *VP3* to be selected (i.e., *checkCardinality* returns $\{\{r1, r2\}\}$ in Line 25). *r3* complies with the cardinality constraint in *VP7*. *checkDecisionRestrictions* returns $\{\{r1, r2\}\}$ for the contradicting restrictions in Fig. 5.3 (Line 27).

5.6 Incremental Reconfiguration of PS Use Case Models

After all the decision changes are made, the PS use case models need to be incrementally reconfigured (*Challenge 2*). The reconfiguration of PS models is implemented as a pipeline (see Fig. 5.4). Configuration decisions are captured in a decision model during the decision-making process. The decision model conforms to a decision metamodel, described in Chapter 4. PUMConf keeps two decision models, i.e., the decision model before changes (*M1* in Fig. 5.4) and the decision model after changes (*M2* in Fig. 5.4). Fig. 5.5 provides the decision metamodel and the two input decision models for the PL use case models in Fig. 3.5 and Table 3.2.

The pipeline takes the decision models, and the PS diagram and specifications as input. The PS models are reconfigured, as output, together with an impact report, i.e., list of reconfigured parts of the PS models. The pipeline has three steps (Fig. 5.4).

In Step 1, *Matching decision model elements*, the structural differencing of *M1* and *M2* is done by looking for the correspondences in *M1* and *M2*. To that end, we devise an algorithm that identifies the matching model elements in *M1* and *M2*. The output of Step 1 is the corresponding elements, representing decisions for the same variations, in *M1* and *M2* (Section 5.6.1).

The decision metamodel in Fig. 5.5(a) includes the main use case elements for which the user makes decisions (i.e., variation point, optional step, optional alternative flow, and variant order). In a variation point, the user selects variant use cases to be included for the product. For PL use case specifications, the user selects optional steps and alternative flows to be included and determines the order of steps (variant order). Therefore, the matching elements in Step 1 are the pairs of variation points and use cases including the variation points, the pairs of use cases and optional alternative flows in the use cases, and the triples of use cases, flows in the use cases, and optional steps in the flows.

In Step 2, *Change calculation*, decision-level changes are identified from the corresponding model elements (see Section 5.6.1). A set of elements in *M1* which does not have a corresponding set of elements in *M2* is considered to be a deleted decision, which we refer to as *DeleteDecision* in the decision-level changes. Analogously, a set of model elements in *M2* which does not have a corresponding set of elements in *M1* is considered to be added (*AddDecision*). Each set of corresponding model elements with non-identical attribute values (see the red-colored attributes in Fig. 5.5(c)) is considered to be a decision-level change of the type *UpdateDecision*. Alternatively, we could record changes during the decision-making process. However, the user might make changes cancelling previous changes or implying some further changes. In such a case, we would have to compute cancelled changes and infer new changes.

In Step 3, *Regeneration of PS models*, the PS use case diagram and specifications are regenerated only for the added, deleted and updated decisions (see Section 5.6.2). For instance, use cases selected in the deleted decisions are removed from the PS use case models, while use cases selected in the added decisions are added in the PS models.

5.6.1 Model Matching and Change Calculation

We devise an algorithm (see Alg. 7) for the first two pipeline steps, *Matching decision model elements* and *Change calculation*, in Fig. 5.4. The algorithm calls some *match* functions (Lines 7-9 in Alg. 7) to identify the corresponding model elements, which represent decisions for the same variations, in the input decision models. The *match* functions implement Step 1 in Fig. 5.4.

- ***matchDiagramDecisions*** returns the set of pairs (*variation point*, *use case*) matching in the decision models (*M1* and *M2*), which are capturing which variation points are included in the use cases involved in diagram decisions,

- ***matchFlowDecisions*** returns the set of pairs (*use case*, *optional alternative flow*) matching in

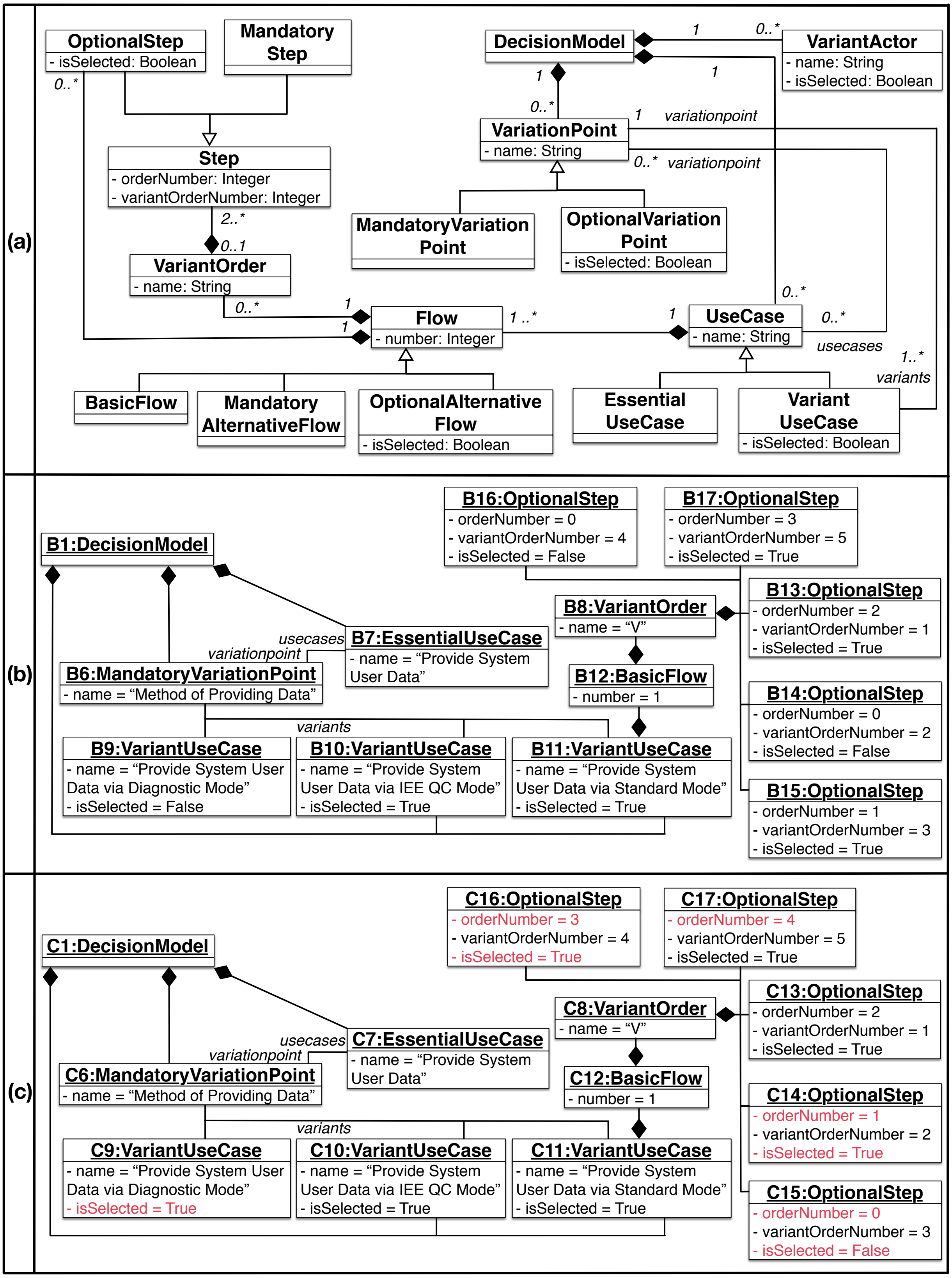


Figure 5.5. (a) Decision Metamodel, (b) Part of the Example Decision Model before Changes (*M1*), and (c) Part of the Example Decision Model after Changes (*M2*)

the input decision models (*M1* and *M2*), which are capturing which optional alternative flows are in the use cases involved in flow decisions,

- ***matchStepDecisions*** returns the set of triples (*use case*, *flow*, *step*) matching in the input decision models (*M1* and *M2*), which are capturing which steps are in the flows of the use cases involved in step decisions.

Alg. 7: Algorithm for Steps 1 and 2 in Fig. 5.4

```
   Inputs : Initial decision model, $M1$, New decision model, $M2$
   Output: Triple of sets of decision-level changes (ADD, DELETE, UPDATE)
1  Let a pair $(vp, uc)$ denote cases where $vp$ is a variation point and $uc$ is a use case including $vp$
2  Let a pair $(uc, fl)$ denote cases where $uc$ is a use case and $fl$ is an optional alternative flow in $uc$
3  Let a triple $(uc, fl, st)$ denote cases where $uc$ is a use case, $fl$ is a flow in $uc$, and $st$ is a step in $fl$
4  Let $U1$ and $U2$ be the sets of $(vp, uc)$ in $M1$ and $M2$
5  Let $F1$ and $F2$ be the sets of $(uc, fl)$ in $M1$ and $M2$
6  Let $S1$ and $S2$ be the sets of $(uc, fl, st)$ in $M1$ and $M2$
7  $U3 \leftarrow$ matchDiagramDecisions($U1$, $U2$)
8  $F3 \leftarrow$ matchFlowDecisions($F1$, $F2$)
9  $S3 \leftarrow$ matchStepDecisions($S1$, $S2$)
10 $DELETE \leftarrow (U1 \setminus U3) \cup (F1 \setminus F3) \cup (S1 \setminus S3)$
11 $ADD \leftarrow (U2 \setminus U3) \cup (F2 \setminus F3) \cup (S2 \setminus S3)$
12 foreach $(k \in (D3 \cap D1))$ do
13     $z \leftarrow$ getMatchingDecision($k$, $U3$)
14     $SUC1 \leftarrow$ getSelectedUseCases($k$, $M1$)
15     $SUC2 \leftarrow$ getSelectedUseCases($z$, $M2$)
16     if $(SUC1 \neq SUC2)$ then
17         $UPDATE \leftarrow UPDATE \cup \{k\}$
18     end if
19 end foreach
20 foreach $(t \in (F3 \cap F1))$ do
21     $y \leftarrow$ getMatchingDecision($t$, $F3$)
22     if $(t.fl.isSelected \neq y.fl.isSelected)$ then
23         $UPDATE \leftarrow UPDATE \cup \{t\}$
24     end if
25 end foreach
26 foreach $(u \in (S3 \cap S1))$ do
27     $m \leftarrow$ getMatchingDecision($u$, $S3$)
28     if $(u.st$ is $OptionalStep)$ and $(u.st.isSelected \neq m.st.isSelected)$ then
29         $UPDATE \leftarrow UPDATE \cup \{u\}$
30     else
31         if $(u.st.orderNumber \neq m.st.orderNumber)$ then
32             $UPDATE \leftarrow UPDATE \cup \{u\}$
33         end if
34     end if
35 end foreach
36 return $(ADD, DELETE, UPDATE)$
```

The corresponding model elements in the decision models in Fig. 5.5(b) and (c) are as follows (Lines 7-9 in Alg. 7):

- For decisions in the variation points, $U3 = \{(B6, B7), (C6, C7)\}$,

- For decisions in the optional alternative flows, $F3 = \emptyset$,

- For decisions in the use case steps, $S3 = \{(B11, B12, B13), (B11, B12, B14), (B11, B12, B15), (B11, B12, B16), (B11, B12, B17), (C11, C12, C13), (C11, C12, C14), (C11, C12, C15), (C11, C12, C16), (C11, C12, C17)\}$.

A variant use case in a variation point (*vp*) may include another variation point (*vp′*). Changing the decision for *vp* may imply another decision to be added or deleted for *vp′*. As part of Step 2, *Change Calculation*, the algorithm first identifies deleted and added diagram decisions by checking the pairs of variation points and use cases which exist only in one of the input decision models (($U1 \setminus U3$) and ($U2 \setminus U3$) in Lines 10-11). Similar checks are done for flow and step decisions in the specifications (Lines 10-11). For the decision models in Fig. 5.5, there is no deleted or added decision (($U1 \setminus U3 = \emptyset$), ($U2 \setminus U3 = \emptyset$), ($F1 \setminus F3 = \emptyset$), ($F2 \setminus F3 = \emptyset$), ($S1 \setminus S3 = \emptyset$), and ($S2 \setminus S3 = \emptyset$)).

The matching pairs of variation points and their including use cases represent decisions for the same variation point (($B6, B7$) and ($C6, C7$) in Fig. 5.5(b) and (c)). If the selected variant use cases for the same variation point are not the same in *M1* and *M2*, the corresponding decision in *M1* is considered as updated in *M2* (Lines 12-19). The variant use case *Provide System User Data via Diagnostic Mode* of the variation point *Method of Providing Data* is unselected in *M1* (*B6*, *B7* and *B9* in Fig. 5.5(b)), but selected in *M2* (*C6*, *C7* and *C9* in Fig. 5.5(c)). The diagram decision for the pair (*B6*, *B7*) in *M1* is identified as updated (Line 17). To identify updated specification decisions, the algorithm compares decisions across *M1* and *M2* that involve optional alternative flows, optional steps and steps with a variant order (Lines 22-24, 28-30 and 31-33). In our example, the triples (*B11*,*B12*,*B14*), (*B11*,*B12*,*B15*), (*B11*,*B12*,*B16*), and (*B11*,*B12*,*B17*) in Fig. 5.5 represent updated decisions.

5.6.2 Regeneration of PS Use Case Models

After all the changes are calculated by matching the corresponding model elements in the input decision models, the parts of PS use case models affected by the changed decisions are automatically regenerated (Step 3 in Fig. 5.4).

Our approach first handles the diagram decision changes to reconfigure the PS use case diagram. For selected variant use cases in the added diagram decisions (i.e., in the pairs (*vp*, *uc*) in *ADD* in Line 36 in Alg. 7), we generate the corresponding use cases and *include* relations in the PS diagram. For selected variant use cases in deleted diagram decisions (i.e., in the pairs (*vp*, *uc*) in *DELETE* in Line 36), we remove the corresponding use cases and *include* relations from the PS diagram. If a selected variant use case is unselected in an updated diagram decision (i.e., in the pairs (*vp*, *uc*) in *UPDATE* in Line 36), we remove the corresponding use case from the PS diagram. For unselected variant use cases which are selected in the updated diagram decisions, the corresponding use cases and *include* relations are added to the PS diagram. Fig. 5.6 gives the regenerated parts of the PS use case diagram in Fig. 4.4 for *M1* and *M2* in Fig. 5.5.

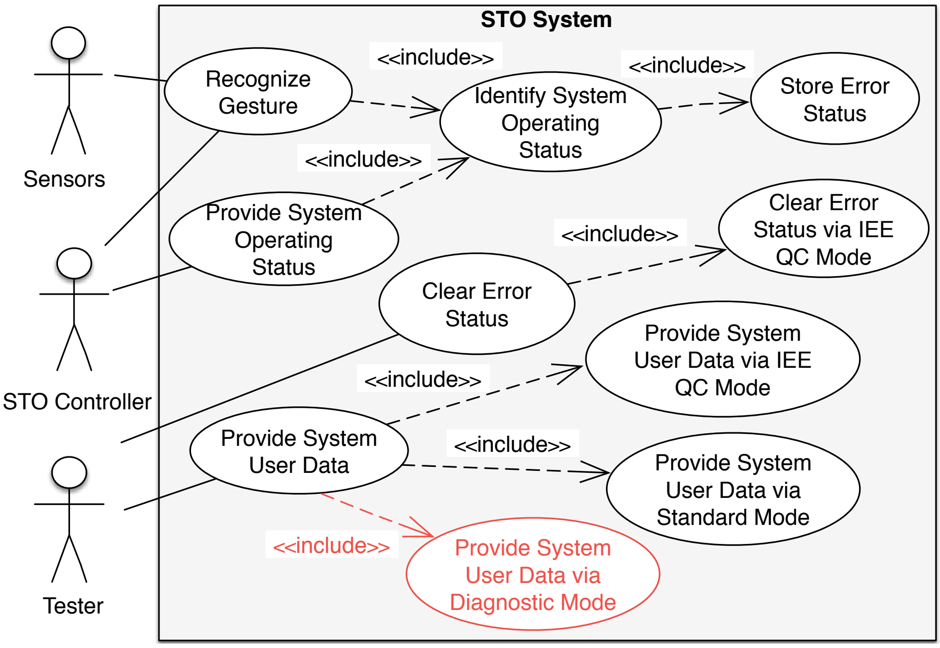

STO System
Sensors
STO Controller
Tester
Recognize Gesture
Identify System Operating Status
Store Error Status
Provide System Operating Status
Clear Error Status via IEE QC Mode
Clear Error Status
Provide System User Data via IEE QC Mode
Provide System User Data
Provide System User Data via Standard Mode
Provide System User Data via Diagnostic Mode
<<include>>
<<include>>
<<include>>
<<include>>
<<include>>
<<include>>
<<include>>

Figure 5.6. Regenerated Product Specific Use Case Diagram

There is no added or deleted diagram decision in *M1* and *M2* in Fig. 5.5. The decision for the variation point *Method of Providing Data* (i.e., $(B6, B7)$ in *UPDATE* in Line 36) is updated by selecting the variant use case *Provide System User Data via Diagnostic Mode*. Only the corresponding use case and its *include* relation are added to the PS use case diagram (red-colored in Fig. 5.6).

Changes for diagram and specification decisions are used to regenerate the PS specifications. For diagram decision changes, we add or delete the corresponding use case specifications. Table 5.4 provides the regenerated parts of the PS specifications in Table 4.2, for *M1* and *M2* in Fig. 5.5.

For the variation point *Method of Providing Data* included by the use case *Provide System User Data* (i.e., $(B6, B7)$), we have one updated diagram decision in which the unselected use case *Provide System User Data via Diagnostic Mode* is selected. The corresponding use case specification is added (Lines 24-29 in Table 5.4). A new specific alternative flow is also generated for the inclusion of the newly selected use case in the specification of the use case *Provide System User Data* (Lines 12-15, red-colored).

The specification decision changes are about selecting optional alternative flows, optional steps and steps with a variant order (e.g., the triples $(B11, B12, B14)$, $(B11, B12, B15)$, $(B11, B12, B16)$, and $(B11, B12, B17)$ in Fig. 5.5(b)). The use case *Provide System User Data via Standard Mode* has two new steps in Lines 19 and 21 in Table 5.4 (i.e., $(B11, B12, B14)$, and $(B11, B12, B16)$ in Fig. 5.5(b)), while one of the steps (red-colored, strikethrough step) is removed (i.e., $(B11, B12, B15)$ in Fig. 5.5(b)). The step number of one of the steps is changed (Line 22, blue-colored) due to the change in the order of the steps with a variant order (i.e., $(B11, B12, B17)$ in Fig. 5.5(b)).

Table 5.4. Reconfigured Product Specific Specifications

USE CASE Provide System User Data
1.1 Basic Flow
1. The tester SENDS the user data request TO the system.
2. The system VALIDATES THAT ‘Precondition of Provide System User Data via Standard Mode’.
3. INCLUDE USE CASE Provide System User Data via Standard Mode.
1.2 Specific Alternative Flow
RFS 2
1. IF ‘Precondition of Provide System User Data via IEE QC Mode’ holds THEN
2. INCLUDE Provide System User Data via IEE QC Mode.
3. ABORT.
4. ENDIF
1.3 Specific Alternative Flow
RFS 2
1. INCLUDE USE CASE Provide System User Data via Diagnostic Mode.
2. ABORT.

USE CASE Provide System User Data via Standard Mode
1.1 Basic Flow
~~1. The system SENDS trace data TO the tester.~~
1. The system SENDS sensor data TO the tester.
2. The system SENDS calibration TO the tester.
3. The system SENDS error data TO the tester.
4. The system SENDS error trace data TO the tester.

USE CASE Provide System User Data via Diagnostic Mode
1.1 Basic Flow
1. The system SENDS the RAM data TO the tester.
2. The system SENDS the NVM data TO the tester.
3. The system SENDS the session response TO the tester.
4. The system SENDS the message length TO the tester.

5.7 Tool Support

We have implemented our change impact analysis approach as an extension of PUMConf (Product line Use case Model Configurator). In Chapter 4, we already the tool architecture including the components for automated configuration of PS use case and domain models. Section 5.7.1 provides the extensions in the layered architecture of the tool for change impact analysis while we describe the tool features with some screenshots in Section 5.7.2. For more details and accessing the tool, see: `https://sites.google.com/site/pumconf/`.

5.7.1 Tool Architecture

Fig. 5.7 shows the tool architecture. It is composed of three layers: (i) the *User Interface (UI) layer*, (ii) the *Application layer*, and (iii) the *Data layer*.

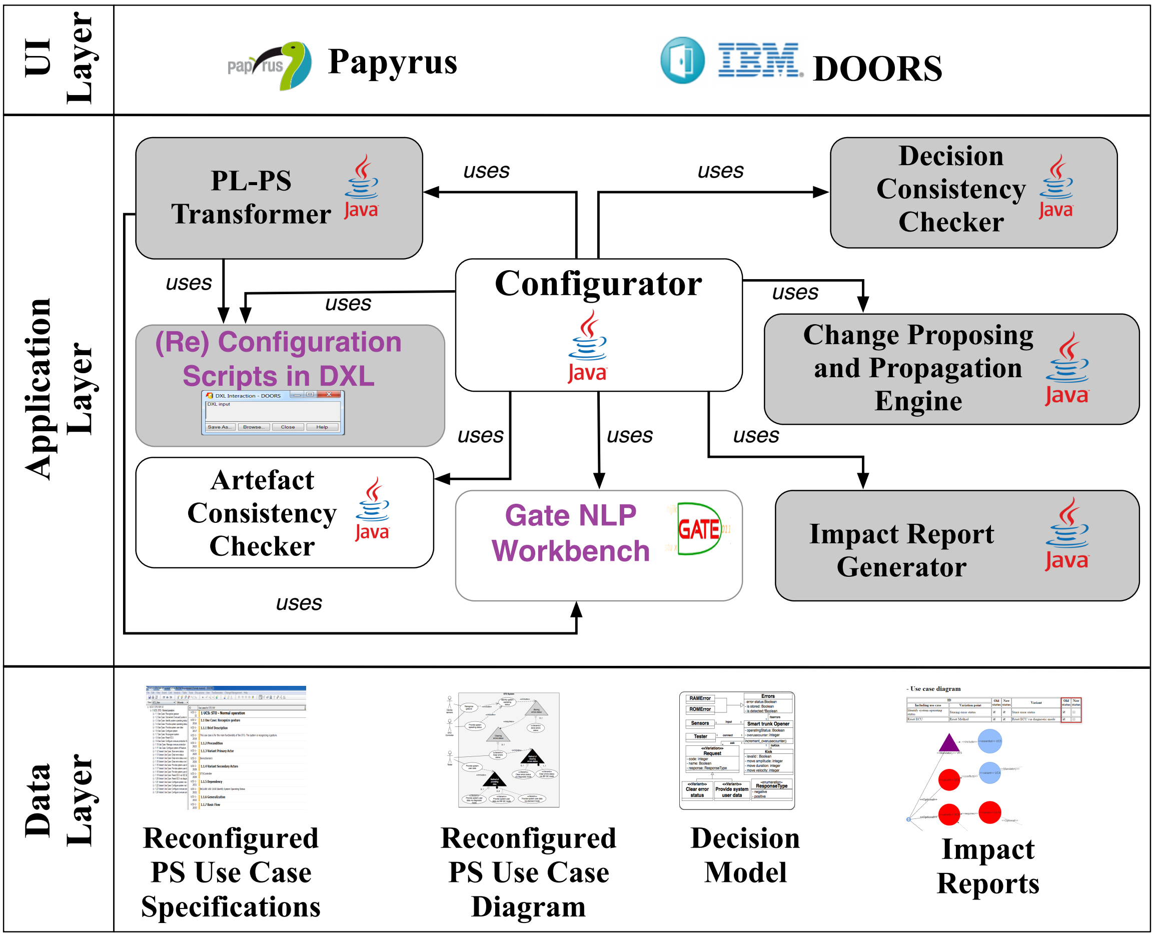


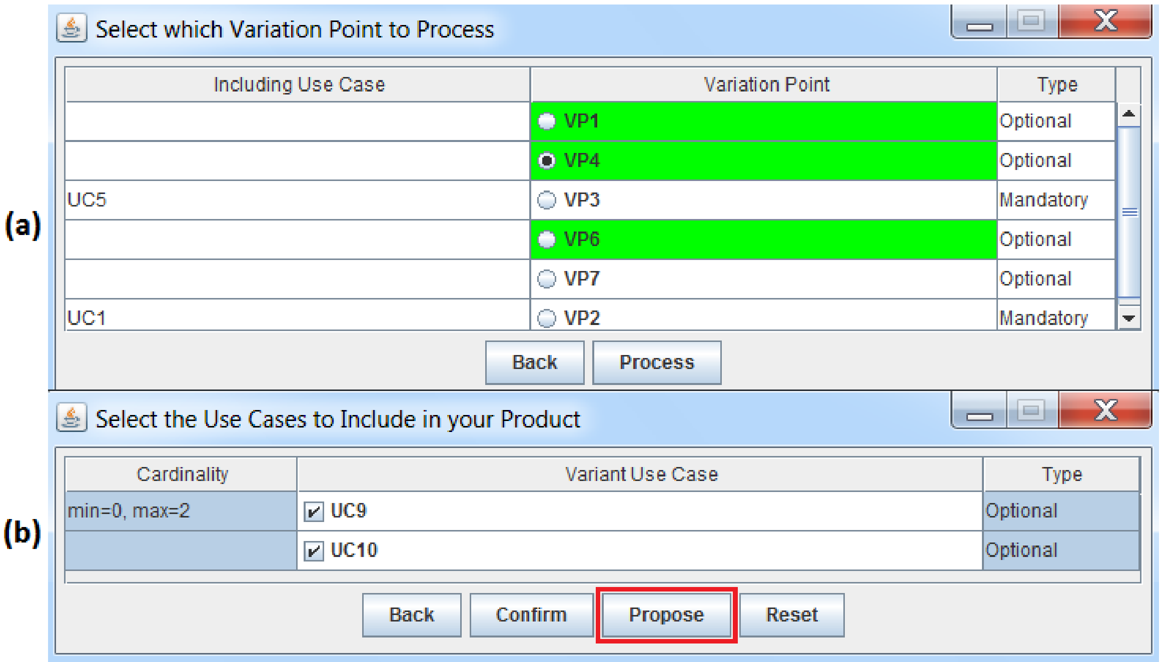
Figure 5.7. Layered Architecture of PUMConf

We briefly introduce each layer and explain the new and extended components, i.e., the gray boxes in Fig. 5.7.

User Interface (UI) Layer. This layer supports creating and viewing PL and PS artifacts, i.e., use case diagrams and specifications. We employ IBM Doors (`www.ibm.com/software/products/ca/en/ratidoor/`) for use case specifications and Papyrus (`https://www.eclipse.org/papyrus/`) for use case diagrams.

Application Layer. With the new and extended components, this layer supports the main activities of our impact analysis approach in Fig. 5.2: *proposing a change*, *identifying the change impact on other decisions*, *applying the proposed change*, and *regenerating PS use case models*.

The *Configurator* component coordinates the other components in the application layer. The *Artifact Consistency Checker* and *Decision Consistency Checker* components were introduced in Chapter 4. The *Artifact Consistency Checker* employs Natural Language Processing (NLP) to check the consistency of the PL use case diagram and the PL use case specifications complying with the RUCM template. To perform NLP, our tool employs the GATE workbench (`http://gate.ac.uk/`), an open source NLP framework. The *Decision Consistency Checker* is extended to support inferring de-


(a)
Select which Variation Point to Process
Back
Process
(b)
Select the Use Cases to Include in your Product
Back
Confirm
Propose
Reset

Including Use Case	Variation Point	Type
	○ VP1	Optional
	● VP4	Optional
UC5	○ VP3	Mandatory
	○ VP6	Optional
	○ VP7	Optional
UC1	○ VP2	Mandatory

Cardinality	Variant Use Case	Type
min=0, max=2	☑ UC9	Optional
	☑ UC10	Optional

Figure 5.8. PUMConf's User Interface for Proposing a Diagram Decision Change

cision restrictions and checking their consistency as part of our impact analysis approach. The *PL-PS Transformer* component annotates the use case specifications using NLP to automatically generate PS use case specifications. It is extended with the pipeline in Fig. 5.4 to incrementally regenerate PS models. It uses scripts written in the Doors eXtension Language (DXL) to automatically (re)configure PS use case specifications. The DXL scripts are also used to load the (re)configured use case specifications into Doors.

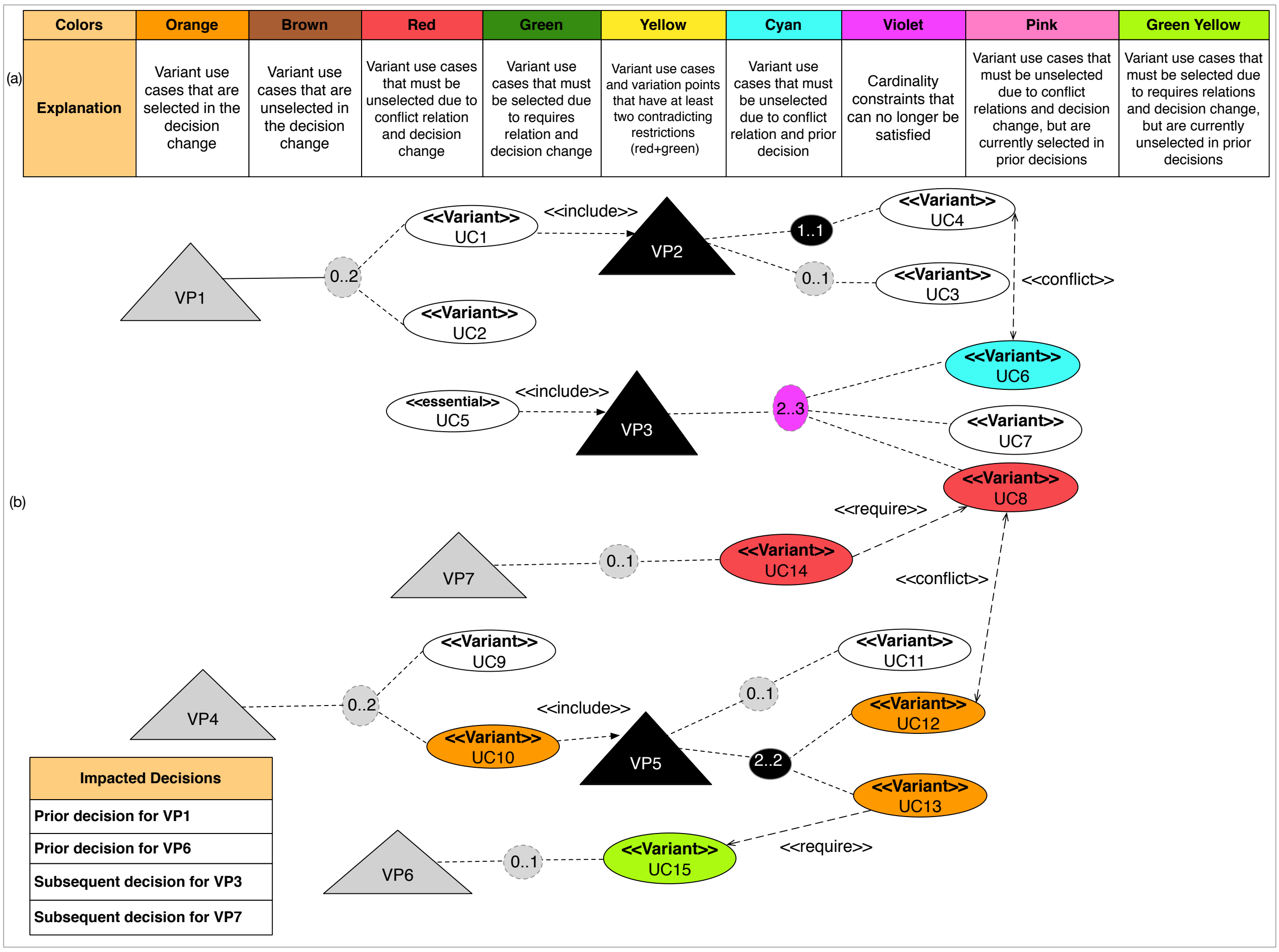
We further implemented some new components: *Change Proposing and Propagation Engine* and *Impact Report Generator*. The *Change Proposing and Propagation Engine* supports proposing a decision change and applying the proposed change while the *Impact Report Generator* generates the impact analysis reports.

Data Layer. The PL and PS use case specifications are stored in the native IBM DOORS format while the PL and PS use case diagrams are stored as UML models. The decision models are saved in Ecore [Ecl, 2018]. We generate the impact reports as html pages.

5.7.2 Tool Features

We describe the main features of our tool: *proposing a decision change*, *identifying the change impact on other decisions*, *applying the proposed change*, and *incrementally reconfiguring PS use case models.*

Proposing a change. This feature supports Step 1, *Propose a Change for a Decision*, in Fig. 5.2.



(a)

Colors	Orange	Brown	Red	Green	Yellow	Cyan	Violet	Pink	Green Yellow
Explanation	Variant use cases that are selected in the decision change	Variant use cases that are unselected in the decision change	Variant use cases that must be unselected due to conflict relation and decision change	Variant use cases that must be selected due to requires relation and decision change	Variant use cases and variation points that have at least two contradicting restrictions (red+green)	Variant use cases that must be unselected due to conflict relation and prior decision	Cardinality constraints that can no longer be satisfied	Variant use cases that must be unselected due to conflict relations and decision change, but are currently selected in prior decisions	Variant use cases that must be selected due to requires relations and decision change, but are currently unselected in prior decisions

(b)

Impacted Decisions
Prior decision for VP1
Prior decision for VP6
Subsequent decision for VP3
Subsequent decision for VP7

Figure 5.9. PUMConf's User Interface for Displaying the Change Impact of Diagram Decision Changes on Other Diagram Decisions

Before applying the change, the analyst proposes the decision change to determine the change impact on other diagram decisions. In Fig. 5.8, the analyst decides to change the decision for the variation point *VP4* (Fig. 5.8(a)) and proposes selecting the unselected *UC10* (Fig. 5.8(b)).

Identifying the change impact on other decisions. For Step 2, *Identify the Change Impact on Other Decisions*, in Fig. 5.2, the tool automatically identifies the impact of the diagram decision changes on prior and subsequent diagram decisions. Once the analyst proposes the change, the tool provides an impact report documenting the impacted decisions along with an explanation for such impact.

Fig. 5.9 shows the impact report for the example change in Fig. 5.3, i.e., selecting the unselected *UC10* in *VP4*. We use various colors, with a legend, on variant use cases and variation points to explain the impacted decisions with the reason of the impact. When the analyst selects the unselected *UC10*, *UC12* and *UC13* are automatically selected (i.e., the orange variant use cases in Fig. 5.9(b)). The prior decision for *VP6* is impacted because *UC15* that is unselected is required by *UC13* which was selected after the change (i.e., the green yellow variant use case in Fig. 5.9(b)). The subsequent decisions for *VP3* and *VP7* are impacted because *UC8* in *VP3* and *UC14* in *VP7* are restricted by the changed decision (i.e., the red variant use cases in Fig. 5.9(b)). The prior decision for *VP1* is yet another impacted decision because of the cardinality constraint in *VP3* (i.e., the violet cardinality constraint in Fig. 5.9(b)). The cardinality constraint can no longer be satisfied with the restriction for *UC8* derived from the changed decision (i.e., the red *UC8* in Fig. 5.9(b)) and with the restriction for *UC6* derived from the prior decision for *VP1* (i.e., the cyan *UC6* in Fig. 5.9(b)).

Applying the proposed change. This feature supports Step 3, *Apply the Proposed Change*, in Fig. 5.2. After evaluating the impact of the proposed change, the analyst decides whether to apply the proposed change on the corresponding decision.

Incrementally reconfiguring PS use case models. This feature supports Step 4, *Regenerate Product Specific Use Case Models*, in Fig. 5.2. Once all the required changes are made, the tool automatically and incrementally regenerates the PS models corresponding to the changed decisions.

5.8 Evaluation

In this section, we evaluate our change impact analysis approach via reporting on (i) the results of a questionnaire survey at IEE aiming at investigating how the approach is perceived to address the challenges listed in Section 5.2 (Section 5.8.1), (ii) discussions with the IEE engineers to gather qualitative insights into the benefits and challenges of applying the approach in an industrial setting (Section 5.8.2), and (iii) an industrial case study, i.e., STO, to demonstrate the feasibility of the incremental reconfiguration of PS use case models (Section 5.8.3) for a representative system.

5.8.1 Questionnaire Study

We conducted a questionnaire study to evaluate, based on the viewpoints of experienced IEE engineers, how well our change impact analysis approach addresses the challenges that we reported in Section 5.2. The study is described and reported according to the template provided by Wohlin et al. [Wohlin et al., 2012].

5.8.1.1 Planning and Design

To evaluate the output of our impact analysis approach in light of the challenges we identified earlier, we had semi-structured interviews with seven participants holding various roles at IEE: software development manager, software team leader, software engineer, system engineer, hardware development engineer, and embedded software engineer. They all had substantial industry experience, ranging from seven to thirty years. All participants, except the hardware development engineer, had previous experience with use case-driven development and modeling. The interview was preceded by presentations illustrating the background approaches (i.e., the PL use case modeling method in

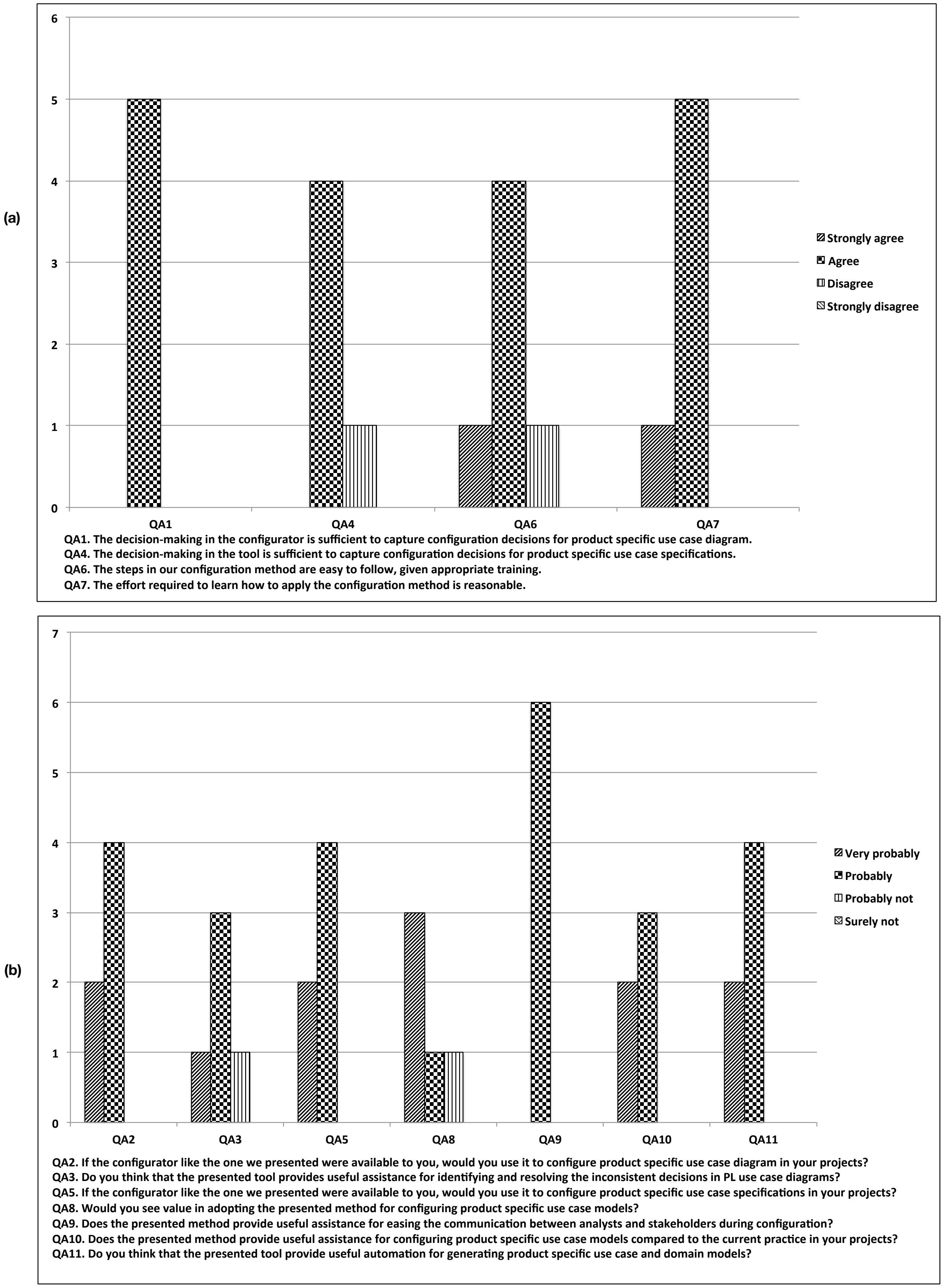
Chapter 3 and the use case-driven configuration approach in Chapter 4), our change impact analysis approach, a tool demo, and some detailed examples from STO. Interactive training sessions also took place which included questions posed to the participants about the example models and ensured that participants had reached a minimal level of understanding. We then organized three hands-on sessions in which the participants could apply the configuration and the change impact analysis approaches in a realistic setting, followed by the structured interviews and data collection. In the first hands-on session, the participants were asked to make configuration decisions and resolve conflicting decisions using the guidance provided by PUMConf to generate PS use case models from the sample PL use case diagram and specifications. In the second hands-on session, they used the impact analysis results provided by PUMConf to identify the impact of the proposed decision changes on prior and subsequent decisions in PL use case diagrams. In the third session, the participants used PUMConf to incrementally reconfigure PS use case models based on the changed decisions.

To capture the perception of the IEE engineers participating in the interviews, regarding the potential benefits of our impact analysis approach and how it addresses the targeted challenges, we handed out two questionnaires including questions to be answered according to two Likert scales [Oppenheim, 2005] (i.e., agreement and probability). The questionnaires were structured for the participants to assess both our configurator and our change impact analysis approach in terms of adoption effort, correctness, comparison with current practice, and tool support. The participants were also encouraged to provide open, written comments.

5.8.1.2 Results and Analysis

We solicited the opinions of the participants using two questionnaires named *QA* and *QB* (see Fig. 5.10 and Fig. 5.11). The objective of the questionnaire *QA* was to evaluate our use case-driven configuration approach and its tool support. We needed to know how well the participants understood and assessed the configuration approach before receiving their feedback about our impact analysis approach, which builds on it. Fig. 5.10(a) and (b) depict the questions in *QA* and the participants' answers. The questions of *QA* were divided into three parts: (1) configuration of PS use case diagrams (*QA*1, *QA*2 and *QA*3), (2) configuration of PS use case specifications (*QA*4 and *QA*5), and (3) the overall configuration approach and its tool support (from *QA*6 to *QA*11).

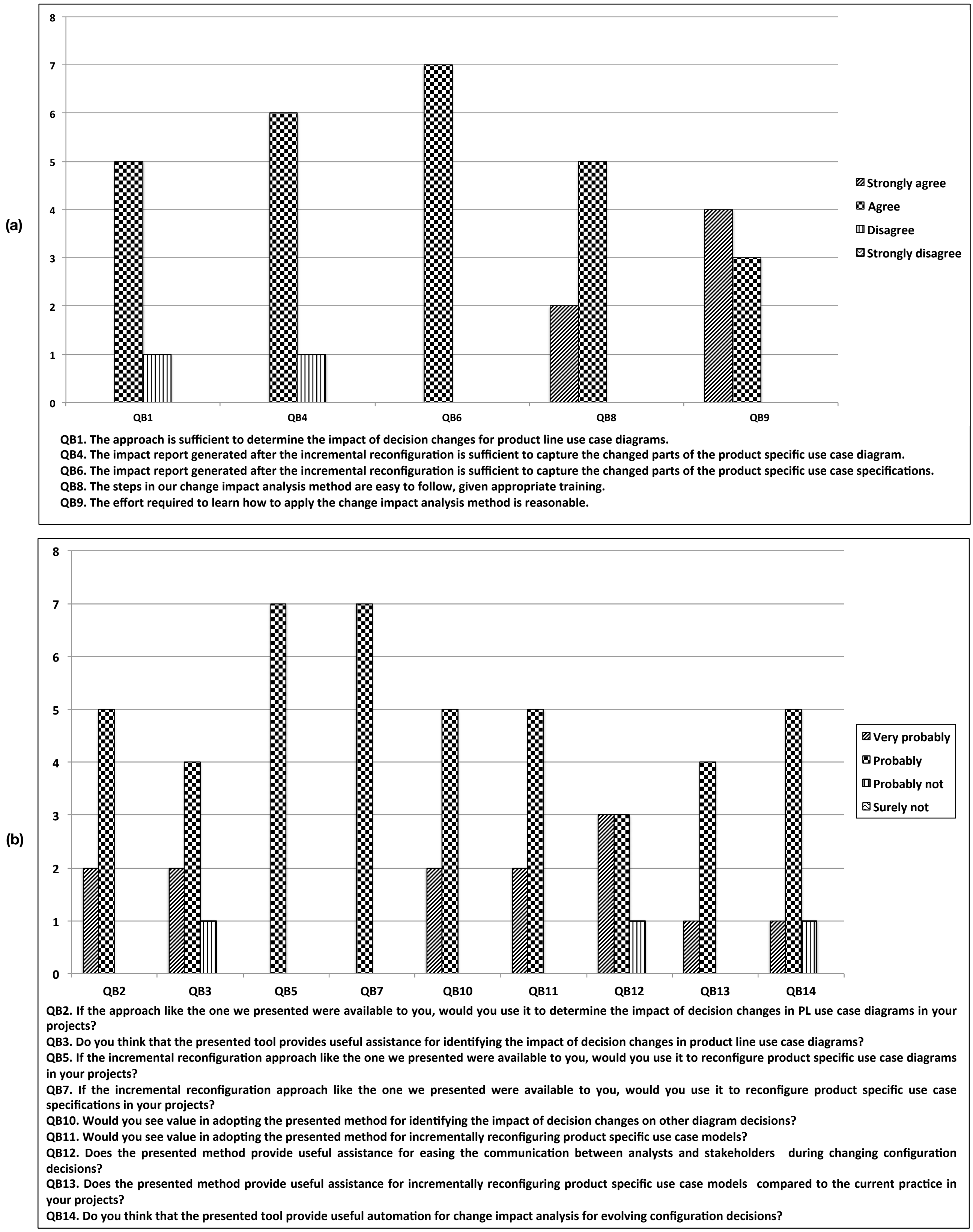
All participants, except two, agreed that our configurator is adequate and practical to capture configuration decisions for PS use case models (*QA*1 and *QA*4). Further, these participants expressed their willingness to use our tool for automatically configuring PS models in their projects (*QA*2 and *QA*5). The two participants who did not *agree* on *QA*4 stated that they need to gather more experience on various product line projects to be able to provide a precise judgment about the configurator. We note that one of those participants *disagreed* whereas the second one left the questions (from *QA*1 to *QA*6) unanswered. The former was the HW engineer, with no initial use case modeling experience, and the latter was the system engineer. In short, these two participants were the ones with the least



QA1. The decision-making in the configurator is sufficient to capture configuration decisions for product specific use case diagram.
QA4. The decision-making in the tool is sufficient to capture configuration decisions for product specific use case specifications.
QA6. The steps in our configuration method are easy to follow, given appropriate training.
QA7. The effort required to learn how to apply the configuration method is reasonable.

QA2. If the configurator like the one we presented were available to you, would you use it to configure product specific use case diagram in your projects?
QA3. Do you think that the presented tool provides useful assistance for identifying and resolving the inconsistent decisions in PL use case diagrams?
QA5. If the configurator like the one we presented were available to you, would you use it to configure product specific use case specifications in your projects?
QA8. Would you see value in adopting the presented method for configuring product specific use case models?
QA9. Does the presented method provide useful assistance for easing the communication between analysts and stakeholders during configuration?
QA10. Does the presented method provide useful assistance for configuring product specific use case models compared to the current practice in your projects?
QA11. Do you think that the presented tool provide useful automation for generating product specific use case and domain models?

Figure 5.10. Responses to the Questions Related to the Configuration Approach



QB1. The approach is sufficient to determine the impact of decision changes for product line use case diagrams.
QB4. The impact report generated after the incremental reconfiguration is sufficient to capture the changed parts of the product specific use case diagram.
QB6. The impact report generated after the incremental reconfiguration is sufficient to capture the changed parts of the product specific use case specifications.
QB8. The steps in our change impact analysis method are easy to follow, given appropriate training.
QB9. The effort required to learn how to apply the change impact analysis method is reasonable.

QB2. If the approach like the one we presented were available to you, would you use it to determine the impact of decision changes in PL use case diagrams in your projects?
QB3. Do you think that the presented tool provides useful assistance for identifying the impact of decision changes in product line use case diagrams?
QB5. If the incremental reconfiguration approach like the one we presented were available to you, would you use it to reconfigure product specific use case diagrams in your projects?
QB7. If the incremental reconfiguration approach like the one we presented were available to you, would you use it to reconfigure product specific use case specifications in your projects?
QB10. Would you see value in adopting the presented method for identifying the impact of decision changes on other diagram decisions?
QB11. Would you see value in adopting the presented method for incrementally reconfiguring product specific use case models?
QB12. Does the presented method provide useful assistance for easing the communication between analysts and stakeholders during changing configuration decisions?
QB13. Does the presented method provide useful assistance for incrementally reconfiguring product specific use case models compared to the current practice in your projects?
QB14. Do you think that the presented tool provide useful automation for change impact analysis for evolving configuration decisions?

Figure 5.11. Responses to the Questions Related to the Change Impact Analysis Approach

software background.

Regarding the questions that target the overall approach and its tool support (from *QA*6 to *QA*11), the participants agreed that the effort required to learn and apply our configurator is reasonable (*QA*7). Nevertheless, one participant stated that more training is required to be able to easily follow the configuration steps (*QA*6). All participants except one were interested in using our configurator for managing product lines. The remaining participant, who is a software project manager and was the most experienced, thought that our configurator brings added value only for projects which include significant variability information, e.g., projects with more than 50 variation points (*QA*8). Moreover, the participants agreed that our configurator provides useful assistance for configuring PS use case models, when compared to the current practice in their projects (*QA*10), and ease communication between analysts and stakeholders during configuration (*QA*9).

The objective of the second questionnaire *QB* was to evaluate our change impact analysis approach. Fig. 5.11(a) and (b) depicts the questions and answers for *QB*. *QB* is structured in four parts: (1) identifying the impact of decision changes on other diagram decisions (from *QB*1 to *QB*3), (2) incrementally reconfiguring PS use case diagrams (*QB*4 and *QB*5), (3) incrementally reconfiguring PS use case specifications (*QB*6 and *QB*7), and (4) the overall impact analysis approach and its tool support (from *QB*8 to *QB*14).

All participants, except one, agreed that (1) our approach is sufficient to determine and explain the impact of decision changes for PL use case diagrams (*QB*1) and (2) the impact report generated after the incremental reconfiguration is sufficient to capture the changed parts of the PS use case diagram (*QB*4). The participant who disagreed on *QB*1 and *QB*4 mentioned in his comments that he lacks experience in use case-driven development and modeling and that he is not sufficiently familiar with the tool to provide a precise answer. There was a strong consensus among participants about the value of adopting our change impact analysis approach (*QB*10 and *QB*11) and about the benefits of using it to identify the impact of decision changes and to reconfigure PS models in their projects (*QB*5 and *QB*7). The participants were very positive about the approach in general and were enthusiastic about its capabilities, and most particularly the impact analysis reports provided by the tool. Nevertheless, they mentioned that the user interface needed to be more professional and ergonomic, which was not surprising for a research prototype. This was the main reason for one of the participants to disagree on *QB*3, *QB*12, and *QB*14.

5.8.2 Discussions with the Analysts and Engineers

The questionnaire study had open, written comments under each section, in which the participants could state their opinions in a few sentences about how our impact analysis approach addresses the challenges reported in Section 5.2. As reported in Section 5.8.1, the participants' answers to the questions through Likert scales and their open comments indicate that they see high value in adopting the change impact analysis approach and its tool support in an industrial setting in terms of (1) improving decision making process, (2) increasing reuse, and (3) reducing manual effort during reconfiguration. In order to elaborate over the open comments in the two questionnaires, we organized further discussions with the participants. Based on the initial comments, we identified two aspects to further discuss with the participants: industrial adoption of the approach and its limitations.

5.8.2.1 Industrial Adoption of the Approach

Our impact analysis approach is devised to support the decision-making process in the context of use case-driven configuration. Therefore, it needs to be adopted as part of our configuration approach. In the current practice at IEE, like many other environments, there is no systematic way to (re)configure product-specific use case models and to identify the change impact for evolving decisions for use case models. Although IEE engineers consider that the effort required to learn and apply our configuration and change impact analysis approach is reasonable, they also stated that the costs and benefits of adopting it should be further evaluated. This is, however, a common and general challenge when introducing new practices in software development.

5.8.2.2 Limitations of the Approach

Our change impact analysis approach and its tool support currently have some limitations. First, our approach supports only evolving configuration decisions. However, changes may also occur on variability aspects of PL use case models. For instance, we may introduce a new variation point in the PL use case diagram or we can remove a variant use case for a given variation point. As stated by IEE engineers, it is important to evaluate the impact of PL use case model changes on configuration decisions and on PS use case models. Therefore, our approach needs to be extended for evolving PL use case models. Second, we implemented our approach as part of a prototype tool, PUMConf. The tool has already received positive feedback from IEE engineers but they stated that it needs further improvements in terms of usability. To identify potential usability improvements, we decided to conduct empirical and heuristic evaluations [Nielsen and Molich, 1990] [Nielsen, 1994]. With regards to the empirical evaluation, we plan to perform a user study with IEE engineers, where we will record the end user interaction with the configurator. We plan to perform the heuristic evaluation of the user interfaces according to certain rules, such as those listed in typical guideline documents [Smith and N.Mosier, 1986], by asking users' opinions about possible improvements of PUMConf's user interfaces.

5.8.3 Industrial Case Study

We report our findings about the feasibility of part of our impact analysis approach, i.e., incremental reconfiguration of PS use case models, and its tool support in an industrial context. In order to experiment with our incremental reconfiguration approach in an industrial project, we applied it to the functional requirements of STO.

5.8.3.1 Goal

Our goal was to assess, in an industrial context, the feasibility of using our approach. We assessed whether we could improve reuse and significantly reduce manual effort by preserving unimpacted parts of PS use case models, when possible, and their manually assigned traces.

5.8.3.2 Study Context

STO was selected for the assessment of our approach since it was a relatively new project at IEE with multiple potential customers requiring different features. IEE provided their initial STO documentation, which contained a use case diagram, use case specifications, and supplementary requirements specifications describing non-functional requirements. To model the STO requirements according to our modeling method, PUM, we first examined the initial STO documentation and then worked with IEE engineers to build and iteratively refine our models (see Table 3.4 in Chapter 3).

5.8.3.3 Results and Analysis

By using PUMConf, we, together with the IEE engineers, configured the PS use case models for four products selected among the STO products IEE had already developed (see Chapter 4). The IEE engineers made decisions on the PL models using the guidance provided by PUMConf. Among the four products, we chose one product to be used for reconfiguration of PS models (see Table 4.3 in Chapter 4) because it was the most recent one in the STO product family with a properly documented change history. The IEE engineers identified 36 traces from the PS use case diagram and 278 traces from the PS use case specifications that were directed to other software and hardware specifications as well as to the customers' requirements documents for external systems (see Fig. 5.1 for an example trace). We considered eight change scenarios derived from the change history of the initial STO documentation for the selected product (see Table 5.5).

Some change scenarios contain individual decision changes such as selecting unselected use cases in a variation point, while some others contain a series of individual changes to be applied sequentially (see *S2* and *S4*). For instance, *S2* starts with unselecting *Clear Error Status* in Fig. 3.5, which automatically deletes the decision for the variation point *Method of Clearing Error Status* and implies another decision change, i.e., unselecting *Store Error Status*.

Table 5.6 provides a summary of the reconfiguration of the PS use case models for the change scenarios. After each change scenario, we ran PUMConf and checked the preserved and deleted traces. As discussed, our approach preserves all the traces for the unchanged parts of the PS models, while removing the traces for the deleted parts of the PS models, which must be manually updated. To assess

Table 5.5. Decision Change Scenarios

ID	Change Scenario	Explanation
S1	Update a diagram decision	Unselecting selected use cases
S2	Update and delete diagram decisions	Unselecting selected use cases, removing other decisions
S3	Update a diagram decision	Selecting unselected use cases
S4	Update and add diagram decisions	Selecting unselected use cases, implying other decisions
S5	Update a specification decision	Selecting unselected optional steps
S6	Update a diagram decision	Selecting unselected use cases
S7	Update a diagram decision	Unselecting selected use cases
S8	Update a specification decision	Updating the order of optional steps

the savings in traceability effort while reconfiguring, we looked at the percentages of traces from the use case diagram and the use case specifications that were preserved over all change scenarios. From Table 5.6, we can see that between 73% and 100% (average $\approx$ 96%) of the use case diagram traces were preserved. Similarly, for the use case specifications, trace reuse was between 82% and 100% (average $\approx$ 96%). We can therefore conclude that the proposed incremental reconfiguration of PS use case models leads to significant savings in traceability effort in the context of actual configuration decision changes.

5.8.4 Threats to Validity

The main threat to validity in our evaluation concerns the generalizability of the conclusions we derived from our industrial case study and from the participants' answers in our questionnaire study. To mitigate this threat, we applied our approach to an industrial case study, i.e., STO, that includes nontrivial use cases in an application domain with many potential customers and sources of variability. STO is a relatively simple but typical automotive embedded system. It can be reasonably argued that more complex systems would require more configuration support, not less. Further case studies are nevertheless necessary for improving external validity. The fact that the respondents to our questionnaire were selected to have diverse backgrounds and the consistency observed across the answers we received provide confidence about the generalizability of our conclusions among different project participants. A potential threat to internal validity is that the we have limited domain knowledge and were involved in the modeling and (re)configuration of the use case models we used in our evaluation. To minimize the risks of mistakes, we had many meetings and interviews with domain experts at IEE to verify the correctness and completeness of (1) our PL use case models, (2) the STO configurations, and (3) the output of our change impact analysis approach.

Table 5.6. Summary of the Reconfiguration of the PS Use Case Models for STO

		Decision Change Scenarios							
		S1	S2	S3	S4	S5	S6	S7	S8
PS Model Changes	# of Added UCs	0	0	1	4	0	1	0	0
	# of Deleted UCs	1	4	0	0	0	0	1	0
	# of Added UC Steps	0	0	53	140	3	85	0	0
	# of Deleted UC Steps	53	140	0	0	0	0	103	0
Traces for the PS Use Case Diagram	# of Initial Traces	36	34	25	27	36	36	38	38
	# of Deleted Traces During Reconfiguration	2	9	0	0	0	0	0	0
	# of Manually Added Traces After Reconfiguration	0	0	2	9	0	2	0	0
	# of Preserved Traces	34	25	25	27	36	36	38	38
	% of Preserved Traces	94.4	73.5	100	100	100	100	100	100
Traces for the PS Use Case Specifications	# of Initial Traces	278	265	218	231	278	287	298	278
	# of Deleted Traces During Reconfiguration	13	47	0	0	0	0	20	0
	# of Manually Added Traces After Reconfiguration	0	0	13	47	9	11	0	0
	# of Preserved Traces	265	218	218	231	278	287	278	278
	% of Preserved Traces	95.3	82.2	100	100	100	100	93.2	100

5.9 Conclusion

This chapter presented a change impact analysis approach that supports evolving configuration decisions in product-line (PL) use case models. The approach automatically identifies the impact of decision changes on other decisions in PL use case models, and incrementally reconfigures PS use case diagrams and specifications for evolving decisions.

We aimed to improve the decision making process by informing the analyst about the impact of decision changes and to minimize manual traceability effort by automatically but incrementally reconfiguring the PS use case models, that is to only modify the affected model parts given a decision change and thus preserve as many traceability links as possible to other artifacts.

Our change impact analysis approach is built on the top of the Product line Use case Modeling method in Chapter 3 and the Product line Use case Model Configurator in Chapter 4, and supported by a tool integrated into IBM DOORS. The key characteristics of our tool are (1) the automated identification of the impact of decision changes on prior and subsequent decisions in PL use case models, and (2) the automated incremental reconfiguration of PS models from PL models and evolving configuration decisions. We performed a case study in the context of automotive domain. The results from structured interviews and a questionnaire study with experienced engineers suggest that our approach is practical and beneficial to analyze the impact of decision changes and to incrementally reconfigure PS models in industrial settings.

In this chapter, we answered *Research Question 3 (To what extent and how can we automate the interactive configuration of use case and domain models? How can we support the analysts for making configuration decisions and for generating PS use case and domain models?)*, *Research Question 4 (What are the change scenarios for use case models and system test cases in a product family? What is necessary for these change scenarios to be handled in the configuration process? Which solutions can be used?)*, and *Research Question 5 (How can a change in a configuration decision be propagated to other decisions in PL use case models and to system test cases? How can we support the analysts in performing changes? How can we reconfigure PS use case and domain models for decision changes? How can we select and prioritize system test cases for such changes?)*.

Our approach does not support the evolution of PL use case models. We still need to address and manage changes in variability aspects of PL models such as adding a new variation point in the PL use case diagram. Our plan for the future work is to support change impact analysis for evolving PL use case models to help analysts properly manage changes in such models (see the future research directions in Chapter 7). This work is an intermediate step to achieve our long term objective in this thesis, i.e., change impact analysis and regression test selection in the context of use case-driven development and testing. Chapter 6 provides an automated regression testing approach for system test cases derived from use case models in product families.

Chapter 6

Automated Test Case Classification and Prioritization in Product Lines

In this chapter, we propose, apply, and assess an automated system test case classification and prioritization approach specifically targeting regression testing in the context of the use case development of product families. Our approach is built upon PUM and PUMConf, which we introduced in Chapters 3 and 4, respectively. Our approach provides: (i) automated support to classify, for a new product in a product family, relevant and valid system test cases associated with previous products, and (ii) automated prioritization of system test cases using multiple risk factors such as the fault proneness of requirements and requirements volatility in a product family. Our evaluation was performed in the context of an industrial product family in the automotive domain. Results provide empirical evidence that we propose a practical and beneficial way to classify and prioritize system test cases for industrial product lines.

6.1 Introduction

Chapter 5 proposed and assessed a change impact analysis approach that supports evolving configuration decisions in product line (PL) use case models. The approach automates the identification of decisions impacted by changes in configuration decisions on PL use case models, and incrementally reconfigures PS use case diagrams and specifications for evolving decisions.

In this chapter, we propose, apply and assess a test case classification and prioritization approach, based on our use case-driven modeling and configuration techniques, to support the incremental testing of new products of a product family where requirements are captured as use case specifications. We do not tackle here the problem of automatically generating system test cases for products in a product family. Some existing test case generation approaches [Wang et al., 2015a] [Wang et al., 2015b] [Yue et al., 2015a] can be adapted to the product line context to generate system test cases and their trace links in the presence of PL requirements. In our context, we aim to automate the identification of system test cases from existing products which are impacted by changes in configuration decisions when a new product is configured. Our approach follows the product line testing strategy *incremental testing of product lines* [de Mota Silveira Neto et al., 2011]. After the initial product is tested individually, new test cases might be needed and some of the existing test cases may need to be modified for new products, while some existing test cases are reused as they are. We aim to test new products using test case classification and prioritization based on configuration changes between the previous product(s) and the new product to be tested.

Our approach supports the classification and prioritization of system test cases for new products in a product line. System test cases for a new product are derived by reusing system test cases for previous product(s), and by identifying use case scenarios of the new product that have not been tested so far in the product family. To reuse the existing system test cases, our approach automatically classifies them as *obsolete*, *retestable*, and *reusable*. An *obsolete* test case cannot be executed on the new product as the corresponding use case scenarios are not selected for the new product. A *retestable* test case is still valid but needs to be rerun to determine the possible impact of changes whereas a *reusable* test case is also valid but does not need to be rerun for the new product. We implemented a model differencing pipeline which identifies changes in the decisions made to configure a product (e.g., selecting a variant use case). There are two sets of decisions: (i) decisions made to generate the PS use case specifications for the previous product(s) and (ii) decisions made to generate the PS use case specifications for the new product. Our approach compares the two sets to classify the decisions as *new*, *deleted* and *updated*, and to identify the impacted parts of the use case models of the previous product(s). By using the trace links from the impacted parts of the use case models to the system test cases, we automatically classify the existing system test cases to be reused for testing the new product. In addition, we automatically identify the use case scenarios of the new product that have not been tested before, and provide information on how to modify existing system test cases to cover these new, untested use case scenarios. System test cases are automatically prioritized based on multiple risk factors such as fault-proneness of requirements and requirements volatility in the product line. To this end, we compute a prioritization score for each system test case based on these factors. To support these activities, we extended our tool, *PUMConf*. We have evaluated the effectiveness of the proposed approach by applying it to select and prioritize the test cases of five software products belonging to a product line in the automotive domain. To summarize, the contributions of this chapter are:

- A test case selection and prioritization approach that is specifically tailored to the use case-driven development of product families and that guides engineers in testing new products in a product family;
- A publicly available tool integrated with IBM DOORS as a plug-in, which automatically selects and prioritizes system test cases when a new product is configured in a product family;

- An industrial case study demonstrating the applicability and benefits of our test selection and prioritization approach.

In this chapter, we answer *Research Question 5 (How can a change in a configuration decision be propagated to other decisions in PL use case models and to system test cases? How can we support the analysts in performing changes? How can we reconfigure PS use case and domain models for decision changes? How can we select and prioritize system test cases for such changes?)*. With the test case classification and prioritization approach, we address the issues about automated identification of change impact on system test cases, automated test case classification, and automated test case prioritization in product lines.

This chapter is structured as follows. Section 6.2 introduces the industrial context of our case study to illustrate the practical motivations for our approach. Section 6.3 discusses the related work in light of the industrial needs identified in Section 6.2. In Section 6.4, we provide an overview of the approach. Sections 6.5 and 6.6 provide the details of its core technical parts. In Section 6.7, we present our tool while Section 6.8 reports on our evaluation in an industrial setting, involving an industrial case study, i.e., STO. In Section 6.9, we conclude the chapter.

6.2 Motivation and Context

Our test case classification and prioritization approach is developed as an extension of our configurator, *PUMConf*, in the context of software systems configured for multiple customers, and developed according to a use case-driven development process. In such a context, requirements variability is communicated to customers using an interactive configuration process and an incremental testing strategy is followed for which guidance and automated support are needed. For instance, for each new product in a product family, IEE negotiates with customers how to resolve variation points in requirements or, in other words, how to configure the product line, and then selects and prioritizes, from the existing test suite(s) of the initial/previous product(s), the system test cases to be executed for the new product. In addition, IEE engineers identify new requirements that have not been tested before and existing test cases that need to be modified for the new product.

The current system testing practice at IEE, like in many other environments, is based on *opportunistic reuse of test assets* [de Mota Silveira Neto et al., 2011] (see Fig. 6.1). Product requirements are elicited from an initial customer and documented as a use case diagram and use case specifications. IEE engineers generate system test cases from the use case specifications. For each new customer in the product family, they copy the current models, and negotiate variabilities with the customer to produce a new use case diagram and new use case specifications (see *copy and modify* in Fig. 6.1). As a result of the negotiations, they make changes in the copied use case models (see *modify*). They manually choose and prioritize, from the existing test suite(s) of the previous product(s), test cases

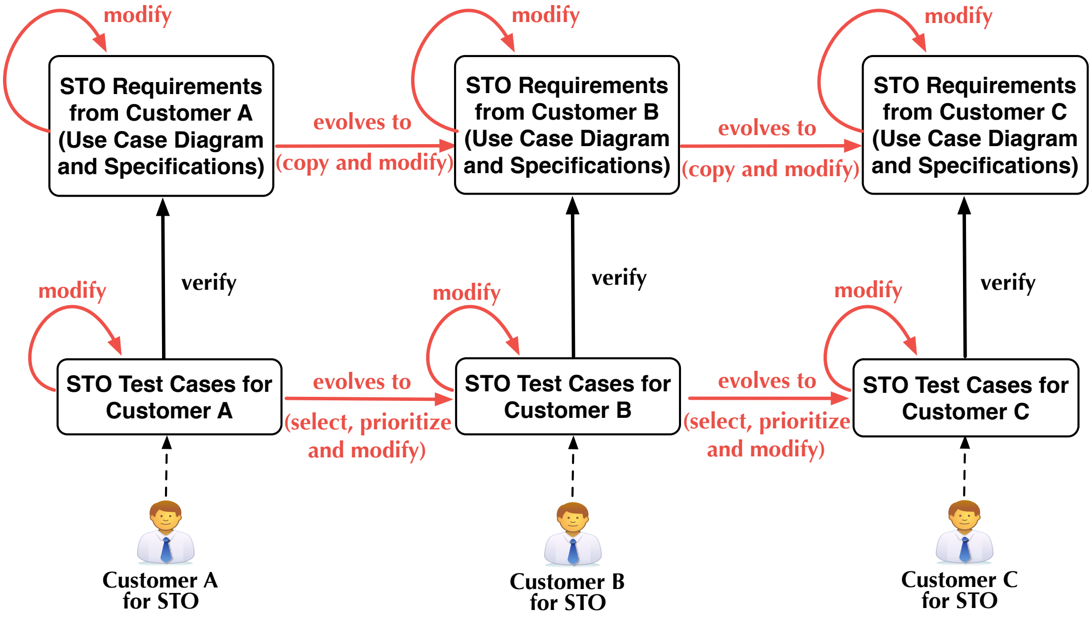

modify
STO Requirements from Customer A (Use Case Diagram and Specifications)
evolves to
(copy and modify)
modify
STO Requirements from Customer B (Use Case Diagram and Specifications)
evolves to
(copy and modify)
modify
STO Requirements from Customer C (Use Case Diagram and Specifications)
modify
verify
modify
verify
modify
verify
STO Test Cases for Customer A
evolves to
(select, prioritize and modify)
STO Test Cases for Customer B
evolves to
(select, prioritize and modify)
STO Test Cases for Customer C
Customer A for STO
Customer B for STO
Customer C for STO

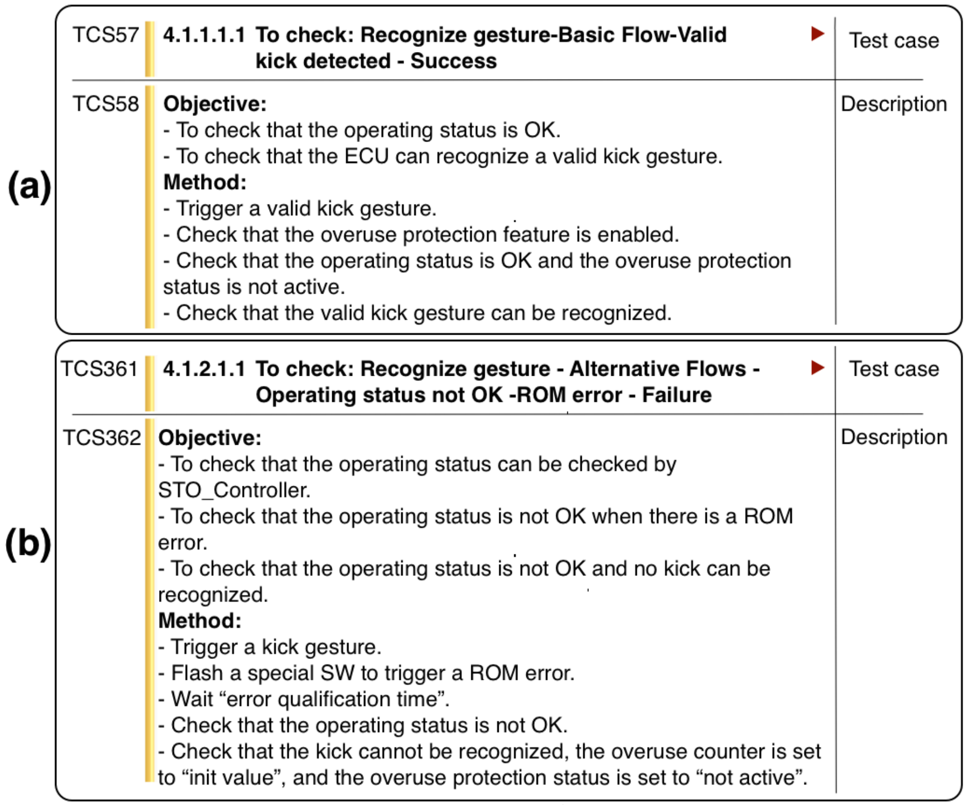
Figure 6.1. Opportunistic Reuse of System Test Cases

that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly in the new product (see *select, prioritize and modify*).

In practice, from a more general standpoint, engineers need more efficient and automated techniques to manage the reuse of common and variable requirements, given as use cases, together with system test cases derived from use cases across products in a product line. This is particularly important in contexts where functional safety standards [RTCA and EUROCAE, 2018] [ISO, 2018] require traceability between requirements and system test cases. In such contexts, traceability [Ramesh and Jarke, 2001] [SWE, 2014] helps guarantee that test cases properly cover all requirements, a very important objective in the standards systems need to comply with. In Chapters 3 and 4, we presented PUM which addresses the reuse of common and variable requirements across products, and PUMConf which supports the automated generation of product specific use case diagrams and specifications. In this chapter, we address the problem of providing automated support for the selection and prioritization of system test cases derived from use cases in a product family.

We identify two challenges that need to be carefully considered when deciding about which system test cases to run on a new product in a product family:

***Challenge 1*: Identifying the Impact of Use Case Changes on System Test Cases.** When there is a new customer requiring a new product in a product family, changes are made in the use case specifications for the new product and act as a contract. Since the generated use case models of the products differ, the test suites used to verify the compliance of the products with their specifications also differ. Therefore, for each new product in the product family, the engineer needs to identify (i) changes on the use case models for the new product, (ii) existing system test cases impacted by those changes, and (iii) the part of the updated use case models which has not been tested yet in the product family.



TCS57 **4.1.1.1.1 To check: Recognize gesture-Basic Flow-Valid kick detected - Success** ▶ Test case

TCS58 **Objective:**
- To check that the operating status is OK.
- To check that the ECU can recognize a valid kick gesture.

Method:
- Trigger a valid kick gesture.
- Check that the overuse protection feature is enabled.
- Check that the operating status is OK and the overuse protection status is not active.
- Check that the valid kick gesture can be recognized.

Description

TCS361 **4.1.2.1.1 To check: Recognize gesture - Alternative Flows - Operating status not OK -ROM error - Failure** ▶ Test case

TCS362 **Objective:**
- To check that the operating status can be checked by STO_Controller.
- To check that the operating status is not OK when there is a ROM error.
- To check that the operating status is not OK and no kick can be recognized.

Method:
- Trigger a kick gesture.
- Flash a special SW to trigger a ROM error.
- Wait "error qualification time".
- Check that the operating status is not OK.
- Check that the kick cannot be recognized, the overuse counter is set to "init value", and the overuse protection status is set to "not active".

Description

Figure 6.2. Test Cases derived from (a) the Basic Flow of the Use Case *Recognize Gesture* in Table 6.1 and (b) the First Specific Alternative Flow of the Same Use Case

Let us consider the two system test cases in Fig. 6.2(a) and (b), which are derived from the use case *Recognize Gesture* in Table 6.1. Fig. 6.2(a) covers the happy path scenario (the basic flow); Fig. 6.2(b) covers the scenario including the alternative flow *SAF1* (Lines 8-10 in Table 6.1) in which the system aborts because of the invalid operating status. For a new STO product, the engineer decides to cover the scenario in which the voltage fluctuation is checked and detected (see the optional bounded alternative flow in Lines 8-12 in Table 3.2). When she changes the configuration for that scenario, the bounded alternative flow is added in the PS use case specification of *Recognize Gesture*. She needs to check (1) if the two test cases verifying *Recognize Gesture* are invalid because they exercise execution scenarios that are impossible due to the new bounded alternative flow, (2) if they need to be re-executed because they still exercise scenarios in the new product which have been impacted by the new alternative flow, or (3) if it is not necessary to re-execute them because the new bounded alternative flow does not have any impact on the scenarios they cover. Further, she needs to derive new test cases to cover the voltage fluctuation scenario that is not covered by the existing test cases.

Challenge 2: **Requirements-based Prioritization of System Test Cases based on Multiple Risk Factors.** Multiple risk factors (such as requirements volatility in the product line, implemen-

Table 6.1. Some of the Generated Product Specific Use Case Specifications for STO

1	**USE CASE** Recognize Gesture
2	**1.1 Basic Flow (BF)**
3	1. The system REQUESTS the move capacitance FROM the sensors.
4	2. INCLUDE USE CASE Identify System Operating Status.
5	3. The system VALIDATES THAT the operating status is valid.
6	4. The system VALIDATES THAT the movement is a valid kick.
7	5. The system SENDS the valid kick status TO the STO Controller.
8	**1.2 Specific Alternative Flow (SAF1)**
9	RFS 3
10	1. ABORT.
11	**1.3 Specific Alternative Flow (SAF2)**
12	RFS 4
13	1. The system increments the OveruseCounter by the increment step.
14	2. ABORT.
15	
16	**USE CASE** Identify System Operating Status
17	**1.1 Basic Flow (BF)**
18	1. The system VALIDATES THAT the watchdog reset is valid.
19	2. The system VALIDATES THAT the RAM is valid.
20	3. The system VALIDATES THAT the sensors are valid.
21	4. The system VALIDATES THAT there is no error detected.
22	**1.5 Specific Alternative Flow (SAF4)**
23	RFS 4
24	1. INCLUDE USE CASE Store Error Status.
25	2. ABORT.
26	
27	**USE CASE** Provide System User Data
28	**1.1 Basic Flow (BF)**
29	1. The tester SENDS the system user data request TO the system.
30	2. The system VALIDATES THAT 'Precondition of Provide System User Data via Standard Mode'.
31	3. INCLUDE USE CASE Provide System User Data via Standard Mode.
32	**1.2 Specific Alternative Flow (SAF1)**
33	RFS 2
34	1. INCLUDE USE CASE Provide System User Data via IEE QC Mode.
35	2. ABORT.
36	
37	**USE CASE** Provide System User Data via Standard Mode
38	**1.1 Basic Flow (BF)**
39	1. The system SENDS the trace data TO the tester.
40	2. The system SENDS the calibration data TO the tester.
41	3. The system SENDS the error trace data TO the tester.

tation complexity of requirements, and fault-proneness of requirements) may have to be considered while system test cases are prioritized for each new product in a product line. For instance, changing requirements, i.e., evolving configuration decisions in the context of automated configuration, cause changes in the design and implementation of the product, and thus increases the likelihood of introducing faults. It may also be desirable to rank higher system test cases for more complex requirements in case the system testing process should be stopped due to deadlines or budget restrictions. These factors may have varying importance for test case prioritization in different product lines due to technical and organizational factors. Therefore, the changing importance of risk factors on test case prioritization should be accounted for in each product line.

In the remainder of this chapter, we focus on how to best address these challenges in a practical manner, in the context of use case-driven development, while relying on PUM for modeling PL use cases, and on PUMConf for the configuration of PS use case models.

6.3 Related Work

We cover the related work across three categories.

Testing of Product Lines. Various product line testing approaches have been proposed in the literature [do Carmo Machado et al., 2014] [de Mota Silveira Neto et al., 2011] [Lee et al., 2012] [Engstrom and Runeson, 2011] [Runeson and Engström, 2012] [Oster et al., 2011] [Tevanlinna et al., 2004] [Johansen et al., 2011]. Neto et al. [de Mota Silveira Neto et al., 2011] present a comprehensive survey of product line testing strategies, i.e., *testing product by product*, *opportunistic reuse of test assets*, *design test assets for reuse*, *division of responsibilities*, and *incremental testing of product lines*. The strategy *testing product by product* ignores the benefits of reusing test cases developed for previous products, while the strategy *opportunistic reuse of test assets* focusses on the reuse of test assets across products without considering any systematic reuse method. The strategy *design test assets for reuse* enforces the creation of test assets early in product line development, under the assumption that product lines and configuration choices are exhaustively modeled before the release of any product. This assumption does not hold when product lines and configuration choices are refined during product configuration, which is a common industry practice. The strategy *division of responsibilities* is about defining testing phases that facilitate test reuse. Our approach follows the strategy *incremental testing of product lines* employing regression testing techniques, i.e., test case selection and prioritization. It is the first to support incremental testing of product lines through test case selection and prioritization for use case-driven development.

Product line testing covers two separate but closely related test engineering activities: domain testing and application testing. Domain testing validates and verifies reusable components in a product line while application testing validates and verifies a product in the product line against its specification. Our approach currently supports application testing, but can be employed in the context of domain testing. For each new product, our approach can be used to classify and prioritize domain test cases derived from PL use case models.

There are various product line testing approaches that support test case generation and execution in product lines (e.g., [Nebut et al., 2006] [Reuys et al., 2006] [Bertolino et al., 2006] [Kamsties et al., 2004] [Geppert et al., 2014] [McGregor, 2001] [Uzuncaova et al., 2010] [Uzuncaova et al., 2008] [Arrieta et al., 2017]). Some of them generate system test cases from use case models in a product family. However, they require detailed behavioural models (e.g., sequence or activity diagrams) which engineers tend to avoid because of the costs related to their development and maintenance. Among these approaches, the main work is that of Reuys et al. [Reuys et al., 2005] [Reuys et al., 2006] [Pohl and Metzger, 2006] [Kamsties et al., 2004], i.e., ScenTED, which is based on the systematic refinement of PL use case scenarios to PL system and integration test scenarios. ScenTED requires activity diagrams capturing activities described in use case specifications together variants of the product family. Extensions of ScenTED include the ScenTED-DF approach [Stricker et al., 2010] which relies on data-flow analysis to avoid redundant execution of test cases derived with ScenTED. A methodology that does not rely on detailed behavioural models is PLUTO (Product Lines Use Case Test Optimization) [Bertolino et al., 2006] [Bertolino and Gnesi, 2003]. PLUTO automatically derives test scenarios from PL use cases with some special tags for variability, but executable system test cases need to be manually derived from test scenarios.

Our approach complements the test generation approaches mentioned above. Not all generated system test cases need to be executed for new products since some of them have already been successfully tested for previous products. UMTG [Wang et al., 2015a] [Wang et al., 2015b] is a promising test generation approach that can be integrated into our approach. It generates system test cases from PS use case specifications in RUCM and from a domain model (class diagram). For a new product in the product family, PUMConf can be used to automatically generate PS use case specifications, which are later taken as input by UMTG to generate system test cases. Our approach can classify and prioritize these test cases for the new product.

Test Case Classification and Selection. When defining a product in a product family for a new customer, there is a need not only for testing the changed parts of the product but also for testing other parts for regression. As the product grows, not all test cases can be rerun for regression due to limited resources. Test case selection is a strategy commonly adopted by regression testing techniques to reduce testing costs [Engstrom, 2010] [Yoo and Harman, 2012] [Do, 2016]. Regression test selection techniques aim to reduce testing costs by selecting a subset of test cases from an existing test suite [Rothermel and Harrold, 1996]. Most of them are code-based and use code changes and code coverage information to guide the test selection (e.g., [Kung et al., 1995] [Binkley, 1997] [Rothermel and Harrold, 1997] [Rothermel et al., 2000] [Harrold et al., 2001] [Qu et al., 2011] [Nardo et al., 2015]). Other techniques use different artifacts such as requirements specifications (e.g., [Vaysburg et al., 2002] [Mirarab et al., 2008] [Dukaczewski et al., 2013]), architecture models (e.g., [von Mayrhauser and Zhang, 1999] [Muccini et al., 2006] [Muccini and van der Hoek, 0382] [Muccini, 2007]), or UML diagrams (e.g., [Briand et al., 2009] [Chen et al., 2002] [Hemmati et al., 2010] [Zech et al., 2017]). For instance, Briand et al. [Briand et al., 2009] present an approach for automating regression test selection based on UML diagrams and traceability information linking UML models to test cases. They propose a formal mapping between changes on UML diagrams (i.e., class and sequence diagrams) and a classification of regression test cases into three categories (i.e., *reusable*, *retestable*, and *obsolete*).

The approaches mentioned above require detailed system design artifacts (e.g., finite state machines and UML sequence diagrams), rather than requirements in natural language, such as use cases. Further, they compare a system artifact with its modified version to select test cases from a single test suite in the context of a single system, not in the context of a product line.

There are several product line test case selection approaches [Runeson and Engström, 2012] [Engström, 2013]. Wang et al. [Wang et al., 2016] [Wang et al., 2017] propose a product line test case selection method using feature models. The method works in three steps: (i) software engineers indicate features that need to be tested; (ii) a toolset is used to check the consistency between features included in a program; and (iii) test cases are automatically selected so that all the test cases associated with a feature to be tested will be executed. The main limitation is that all the test cases of the product family need to be derived upfront even if some of them may never be executed. There are other similar approaches suffering from the same limitation [Cabral et al., 2010] [Knapp et al., 2014] [Shurr et al., 2010] [Kahsai et al., 2008]. In contrast, our approach requires that only test cases of the initial product be available in advance.

A test case selection approach that does not require early generation of test cases for the product family is that of Lity et al. [Lity et al., 2016] [Lochau et al., 2014] [Lity et al., 2012], which is based on model slicing for incremental product line testing. Lity et al. apply incremental model slicing to determine the impact of changes on a test model, e.g., finite state machines, and to reason about their potential retest. The approach needs detailed test models, e.g., finite state machines, which rarely exist in contexts where requirements are mostly captured in NL. In addition, Lity et al. do not support the definition of test cases for new requirements while our approach identifies use case scenarios that have not been tested before, and provides information of how to modify existing test cases to cover those new, untested scenarios (*Challenge 1*). Dukaczewski et al. [Dukaczewski et al., 2013] briefly discuss how to apply incremental product line testing strategies to NL requirements. They do not provide any method to model variability in requirements; it is only suggested that a requirement is split into several requirements, one for each possible product variant. Also, there is no reported systematic approach supported by a tool.

Test Case Prioritization. Test case prioritization techniques schedule test cases in an order that increases their effectiveness in meeting some performance goals (e.g., rate of fault detection and number of test cases required to discover all the faults) [Rothermel et al., 2001] [Khatibsyarbini et al., 2018] [Yoo and Harman, 2012]. They mostly use information about previous executions of test cases (e.g., [Wong et al., 1997] [Rothermel et al., 2001] [Li et al., 2007] [Engström et al., 2011] [Gonzales-Sanchez et al., 2011] [Hemmati et al., 2017]), human knowledge (e.g., [Srikanth et al., 2014] [Srikanth and Williams, 2005] [Srikanth et al., 2016] [Srikanth and Banerjee, 2012] [Arafeen and Do, 2013] [Krishnamoorthi and Mary, 2009] [Basanieri et al., 2002]), or a model of the system under test (e.g., [e Zehra Haidry and Miller, 2013] [Kundu et al., 2009] [Tahat et al., 2012] [Korel et al., 2005] [Korel et al., 2008]). For instance, Shrikanth et al. [Srikanth et al., 2005] propose a test case prioritization approach that takes into consideration customer-assigned priorities of requirements, developer-perceived implementation complexity, requirements volatility, and fault proneness of requirements. Tonella et al. [Tonella et al., 2006] propose a test case prioritization technique using user knowledge through a machine learning algorithm (i.e., Case-Based Ranking). Lachmann et al. [Lachmann et al., 2016b] propose another test case prioritization technique for system-level regression testing based on supervised machine learning. In contrast to the aforementioned approaches, we do aim at prioritizing test cases for a new product in a product family, not for the next version of a single system. Our approach considers multiple risk factors in a product line, identifies their impact on the test case prioritization for the previous products in the product line, and prioritizes test cases for a new product accordingly (*Challenge 2*).

There are some other approaches that address product lines (e.g., [Runeson and Engström, 2012] [Engström, 2013] [Al-Hajjaji et al., 2014] [Baller et al., 2014] [Henard et al., 2014] [Ensan et al., 2011] [Devroey et al., 2017] [Devroey et al., 2014] [Al-Hajjaji et al., 2017a] [Al-Hajjaji et al., 2017b] [Lity et al., 2017]). For instance, to increase feature interaction coverage during product-by-product testing, Al-Hajiaji et al. [Al-Hajjaji et al., 2014] [Al-Hajjaji et al., 2016] propose a similarity-based prioritization approach that incrementally selects the most diverse product in terms of features to be tested. Baller et al. [Baller et al., 2014] propose an approach to prioritize products in a product family based on the selection of test suites with regard to cost/profit objectives. The aforementioned techniques prioritize the products to be tested, which is not useful in our context since products are seldom developed in parallel. In contrast, our approach prioritizes the test cases of a new product to support early detection of software faults based on multiple risk factors (*Challenge 2*).

There are search-based approaches for multi-objective test case prioritization in product lines (e.g., [Wang et al., 2014] [Parejo et al., 2016] [Arrieta et al., 2016] [Pradhan et al., 2018] [Arrieta et al., 2019]). For instance, Parejo et al. [Parejo et al., 2016] model test case prioritization as a multi-objective optimization problem and implement a search-based algorithm to solve it based on the NSGA-II evolutionary algorithm. Arrieta et al. [Arrieta et al., 2019] propose another approach that cost-effectively optimizes the test process of product lines. None of these works consider test case classification and NL requirements as a factor to prioritize test cases.

Lachmann et al. [Lachmann et al., 2015] introduce a test case prioritization technique for incremental testing of product lines using delta-oriented architecture models. The differences between products are captured in the form of *deltas* [Clarke et al., 2010], which are modifications between architecture models of products used for integration testing. The proposed approach ranks test cases based on the number of changed elements in the architecture. It is later extended using risk factors [Lachmann et al., 2017] and behavioral knowledge of architecture components [Lachmann et al., 2016a]. The approach proposed by Lachmann et al. requires access to product architecture descriptions and information about component behavior. In contrast, we do not require any design information but relies on natural language requirements specifications, more precisely use case specifications (*Challenge 2*).

6.4 Overview of the Approach

The process in Fig. 6.3 presents an overview of our approach. In Step 1, *Classify system test cases for the new product*, our approach takes as input (i) system test cases, PS use case models, their trace links, and configuration decisions for previous products in the product family, and (ii) PS use case models and configuration decisions for the new product, to classify the system test cases for the new product as *obsolete*, *retestable*, and *reusable*, and to provide information on how to modify obsolete system test cases to cover new, untested use case scenarios (*Challenge 1*).

Step 1 is fully-automated. The classification and modification information in output of this step is for the test engineer to decide which test cases to execute for the new product and which modifications to make on the obsolete test cases to cover untested, new use case scenarios. We give the details of this step in Section 6.5.

In Step 2, *Select and modify system test cases for the new product*, by using the classification information and modification guidelines automatically provided by our approach, the engineer decides which test cases to run for the new product and modifies obsolete test cases to cover untested, new use case scenarios. The activity is not automated because, for the selection of system test cases, the engineer may also need to consider some implementation and hardware changes (e.g., code refactoring and replacing some hardware with less expensive technology) in addition to the classification information provided in Step 2, which is purely based on changes in functional requirements. For instance, a reusable test case might need to be rerun because part of the source code verified by the test case is refactored.

In Step 3, *Prioritize system test cases for the new product*, selected test cases are automatically prioritized based on risk factors including fault proneness of requirements, and requirements volatility (*Challenge 2*). We discuss this step in Section 6.6.

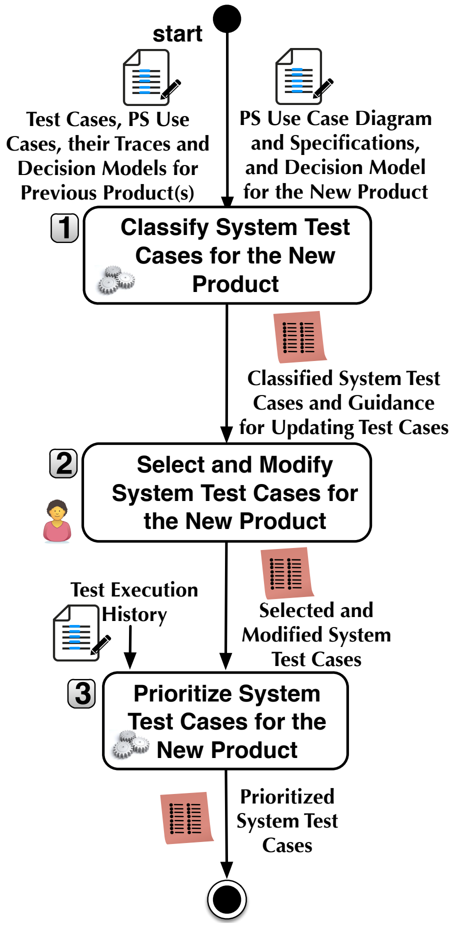

start
Test Cases, PS Use Cases, their Traces and Decision Models for Previous Product(s)
PS Use Case Diagram and Specifications, and Decision Model for the New Product
1
Classify System Test Cases for the New Product
Classified System Test Cases and Guidance for Updating Test Cases
2
Select and Modify System Test Cases for the New Product
Test Execution History
Selected and Modified System Test Cases
3
Prioritize System Test Cases for the New Product
Prioritized System Test Cases

Figure 6.3. Overview of the Approach

6.5 Classification of System Test Cases in a Product Family

The test case classification is implemented as a pipeline (see Fig. 6.4), which takes as input the configuration decisions made for the previous products, the configuration decisions made for the new product, and the previous product's system test cases, trace links, and PS use case models. The pipeline produces an impact report with the list of existing test cases classified.

Configuration decisions are captured in a decision model that is automatically generated by PUMConf during the configuration process. The decision model conforms to the decision metamodel in Fig. 5.5(a). The metamodel includes the main use case elements for which the user makes decisions (i.e., variation points, optional steps, optional alternative flows, and variant orders). PUMConf keeps a decision model for each configuration in the product line. Fig. 6.5 provides two decision models for the PL use case models in Fig. 3.5 and Table 3.2.

The pipeline has four steps (see Fig. 6.4). The first three steps are run for each of the n previous products in the product line, where each one has a decision model M_i with $i = 1..n$. Note that we also employ the first two steps of the pipeline in Fig. 5.4 in Chapter 5. In Step 1, *Matching decision model elements*, our approach automatically runs the structural differencing of M_i and M_{new} by looking for

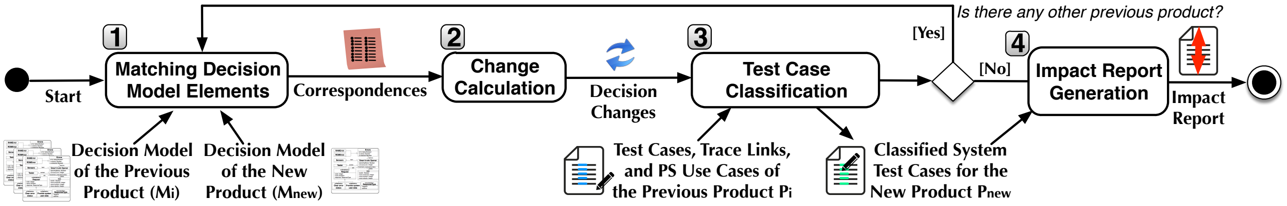

Is there any other previous product?
[Yes]
1
Matching Decision Model Elements
Start
Correspondences
2
Change Calculation
Decision Changes
3
Test Case Classification
[No]
4
Impact Report Generation
Impact Report
Decision Model of the Previous Product (Mi)
Decision Model of the New Product (Mnew)
Test Cases, Trace Links, and PS Use Cases of the Previous Product Pi
Classified System Test Cases for the New Product Pnew

Figure 6.4. Overview of the Model Differencing and Test Case Classification Pipeline

corresponding model elements representing decisions for the same variations (see Section 6.5.1).

In Step 2, *Change calculation*, the approach determines how configuration decisions of the two products differ. Table 5.1 lists the types of decision changes. A decision is represented by means of a n-tuple of model elements in a decision model. A change is of type "Add a decision" when a tuple representing a decision in M_{new} has no matching tuple in M_i. A change is of type "Delete a decision" when a tuple representing a decision in M_i has no matching tuple in M_{new}. A change is of type "Update a Decision" when a tuple representing a decision in M_i has a matching tuple in M_{new} with non-identical attribute values (see the red-colored attributes in Fig. 6.5(c)).

In Step 3, *Test case classification*, the system test cases of the previous products are classified for the new product by using the decision changes obtained from Step 2 and the trace links between the system test cases and the PS use case specifications (see Section 6.5.2). A use case can describe multiple use case scenarios (i.e., sequences of use case steps from the start to the termination of the use case) because of the presence of conditional steps. Each system test case is expected to exercise one use case scenario. For instance, there are three use case scenarios in the use case *Provide System User Data* in Table 6.1. For each use case of the new product, we identify the impact of the decision change(s) on the use case scenarios, i.e., any change in the execution sequence of the use case steps in the scenario.

A system test case is classified in one of three categories: *obsolete*, *retestable* and *reusable*. A test case is obsolete if it exercises an invalid execution sequence of use case steps in the new product. A test case is retestable if it exercises an execution sequence of use case steps that has remained valid in the new product, except for internal steps representing internal system operations (e.g., reset of counters). A test case is reusable if it exercises an execution sequence of use case steps that has remained valid in the new product. The test case categories are mutually exclusive. Use case scenarios of the new product that have not been tested for the previous product are reported as new use case scenarios.

In Step 4, *Impact report generation*, we automatically generate an impact report from the classified test cases of each previous product to enable engineers to select test cases from more than one test suite (see Section 6.5.3). Steps 1, 2 and 3 are the pairwise comparison of each previous product with the new product. If there are multiple previous products ($n > 1$ in Fig. 6.4), test cases of each product are classified separately in n reports in Step 3. We generate an overall impact report that compares

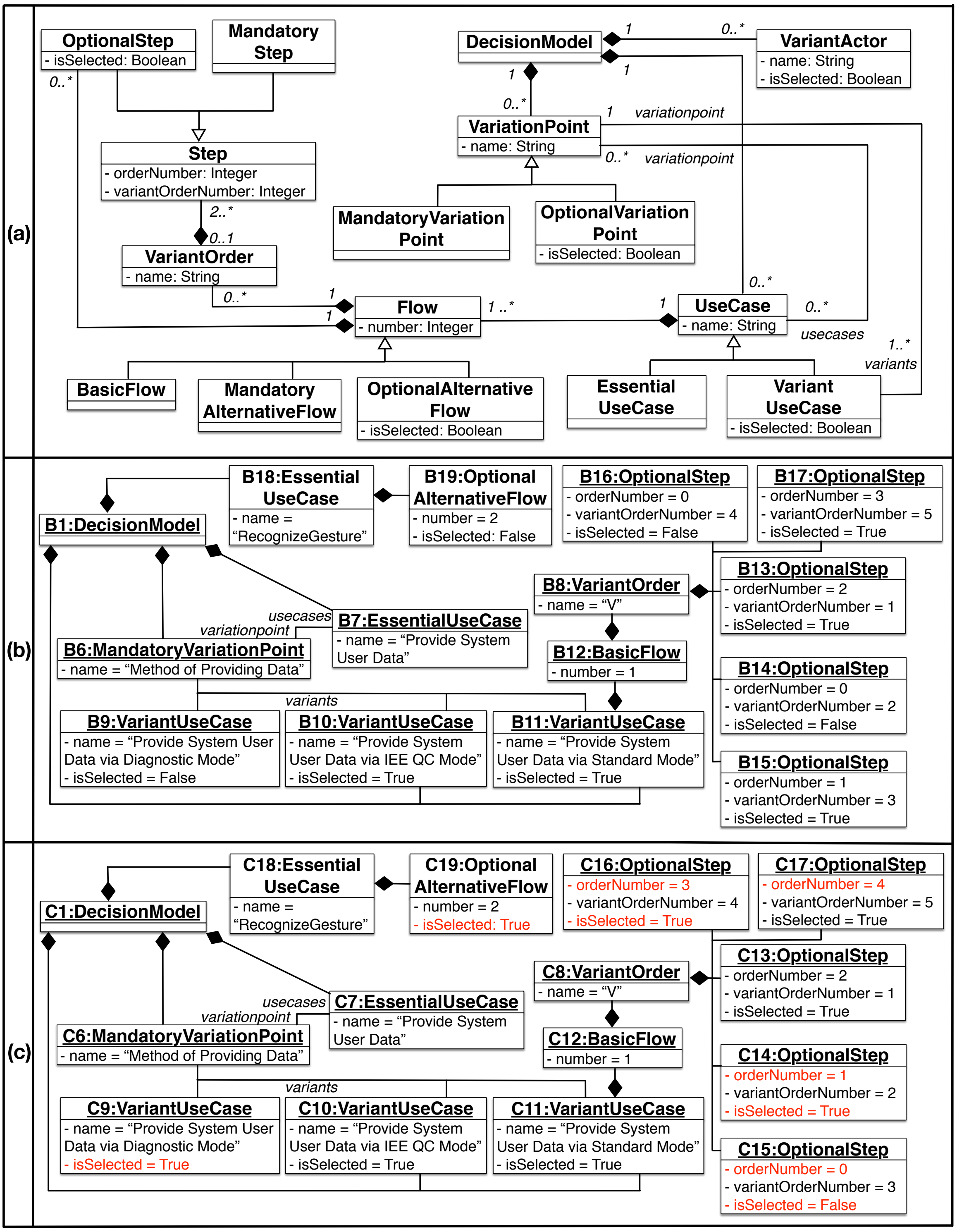


Figure 6.5. (a) Decision Metamodel, (b) Part of the Example Decision Model of the Previous Product (M_i), and (c) Part of the Example Decision Model of the New Product (M_{new})

these n separate reports and lists sets of new scenarios and reusable and retestable test cases for the n previous products.

6.5.1 Steps 1 and 2: Model Matching and Change Calculation

For the first two pipeline steps in Fig. 6.4, we rely on a model matching and change calculation algorithm we devised in Chapter 5. In this section, we provide a brief overview of the two steps and their output for the example decision models in Fig. 6.5(b) and (c).

In Step 1, we identify pairs of decisions in M_i and M_{new} that are made for the same variants. The decision metamodel in Fig. 6.5(a) includes the main use case elements for which the user makes decisions (i.e., variation point, optional step, optional alternative flow, and variant order). In a variation point included by a use case, the user selects variant use cases to be included for the product. For PL use case specifications, the user selects optional steps and alternative flows to be included and determines the order of steps (variant order). Therefore, the matching decisions in Step 1 are the pairs of variation points and use cases including the variation points, the pairs of use cases and optional alternative flows in the use cases, and the triples of use cases, flows in the use cases, and optional steps in the flows. Table 6.2 shows some pairs of decisions in Fig. 6.5(b) and (c). For example, the pairs $\langle B6, B7\rangle$ and $\langle C6, C7\rangle$ represent two decisions for the variation point *Method of Providing Data* included in the use case *Provide System User Data*. The triples $\langle B11, B12, B13\rangle$ and $\langle C11, C12, C13\rangle$ represent two decisions for an optional use case step in the basic flow of the use case *Provide System User Data via Standard Mode*.

Table 6.2. Matching Decisions in M_i and M_{new} in Fig. 6.5

Decisions in M_i	**Decisions in M_{new}**
<B6, B7 >	<C6, C7 >
<B18, B19 >	<C18, C19 >
<B11, B12, B13 >	<C11, C12, C13 >
<B11, B12, B14 >	<C11, C12, C14 >
<B11, B12, B15 >	<C11, C12, C15 >
<B11, B12, B16 >	<C11, C12, C16 >
<B11, B12, B17 >	<C11, C12, C17 >

In Step 2, *Change Calculation*, we first identify deleted and added configuration decisions by checking tuples of model elements in one input decision model which do not have any matching tuples of model elements in another input decision model (M_i and M_{new}). For the decision models in Fig. 6.5, there are no deleted or added decisions. To identify updated decisions, we check tuples of model elements in M_i that have matching tuples of model elements in M_{new} with non-identical attribute values. The matching pairs of variation points and their including use cases represent decisions for the same variation point (e.g., $\langle B6, B7\rangle$ and $\langle C6, C7\rangle$ in Table 6.2). If the selected variant use cases for the same variation point are not the same in M_i and M_{new}, the decision in M_i is considered as updated in M_{new}. We have similar checks for optional steps, optional alternative flows and variant order of steps. For instance, an optional step is selected in the decision represented by the triple $\langle B11, B12, B15\rangle$ in M_i, while the same optional step is unselected in the decision represented by the matching triple

$\langle C11, C12, C15\rangle$ in M_{new}. For the decision models in Fig. 6.5, the decisions represented by $\langle B6, B7\rangle$, $\langle B18, B19\rangle$, $\langle B11, B12, B14\rangle$, $\langle B11, B12B15\rangle$, $\langle B11, B12, B16\rangle$, and $\langle B11, B12, B17\rangle$ are identified as *updated*.

6.5.2 Step 3: Test Case Classification

System test cases of the previous product are automatically classified based on the identified changes (Step 3 in Fig. 6.4). To this end, we devise an algorithm (see Alg. 8) which takes as input a set of use cases (*UC*), the test suite of the previous product (*ts*), and a triple of the sets of configuration changes (*dc*) produced in Step 2. It classifies the test cases and reports use case scenarios of the new product that are not present in the previous product.

Alg. 8: Test Case Classification Algorithm

```
Inputs : Set of use case specifications of the previous product $UC$,
         Test suite of the previous product $ts$,
         Triple of sets of decision-level changes $dc$
         $(ADD, DELETE, UPDATE)$
Output: Quadruple of sets of classified test cases $classified$

Let $OBSOLETE$ be the empty set for obsolete test cases
Let $REUSE$ be the empty set for reusable test cases
Let $RETEST$ be the empty set for retestable test cases
Let $NEW$ be the empty set for new use case scenarios
Let $classified$ be the quadruple (OBSOLETE, REUSE, RETEST, NEW)
foreach ($u \in UC$) do
    if (there is a change in dc for u) then
        $model \leftarrow$ generateUseCaseModel($u$)
        Let $u_{new}$ be a new version of $u$ after the changes in $dc$
        $Scenarios \leftarrow$identifyTestedScenarios($model$, $ts$)
        foreach ($s \in Scenarios$) do
            $T \leftarrow$ retrieveTestCases($s$, $ts$, $Scenarios$)
            $classified \leftarrow classified \cup$ analyzeImpact($s$, $T$, $u_{new}$, $dc$)
        end foreach
    else
        $REUSE \leftarrow REUSE \cup$ retrieveTestCases($u$, $ts$)
    end if
end foreach
$NEW \leftarrow$ filterNewScenarios($NEW$)
return $classified$
```

For each use case in the previous product, we check whether it is impacted by some configuration changes (Lines 6-7 in Alg. 8). If there is no impact, all the system test cases of the use case are classified as *reusable* (Lines 15-17); otherwise, we rely on the function *generateUseCaseModel* (Line 8) to generate a *use case model*, i.e., a model that captures the control flow in the use case. This model is used to identify scenarios that have been tested by one or more test cases (*identifyTestedScenarios* in Line 10). For each scenario verified by a test case (*retrieveTestCases* in Line 12), we rely on the function *analyzeImpact* (Line 13) to determine how decision changes affect the behaviour of the scenario.

In Sections 6.5.2.1, 6.5.2.2, 6.5.2.3 and 6.5.2.4, we give the details of the functions *generateUseCaseModel*, *identifyTestedScenarios*, *retrieveTestCases* and *analyzeImpact*, respectively.

6.5.2.1 Use Case Model Generation

To generate a use case model from a PS use case specification, we rely on a Natural Language Processing (NLP) solution proposed by Wang et al. [Wang et al., 2015a]. It relies on the RUCM keywords and part-of-speech tagging to extract information required to build a use case scenario model. In this section, we briefly describe the metamodel for use case scenario models, shown in Fig. 6.6, and provide an overview of the model generation process.

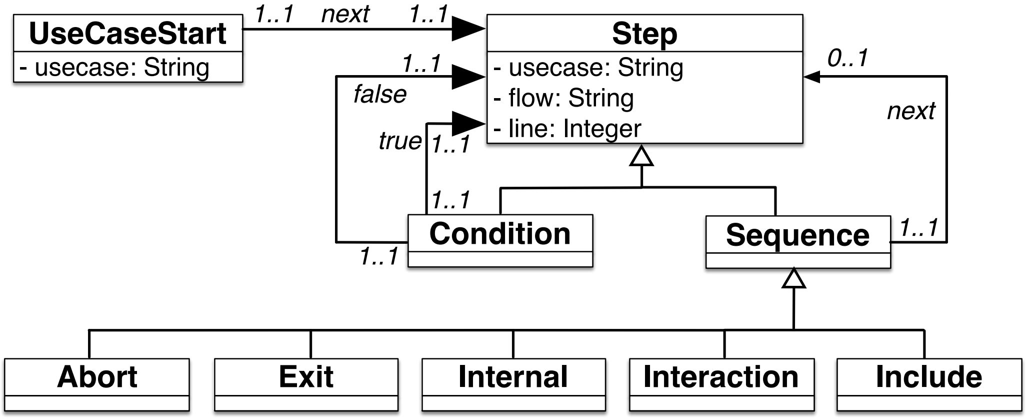


Figure 6.6. Metamodel for Use Case Scenario Models

UseCaseStart represents the beginning of a use case with a precondition and is linked to the first *Step* (i.e., *next* in Fig. 6.6). There are two *Step* subtypes, i.e., *Sequence* and *Condition*. *Sequence* has a single successor, while *Condition* has two successors (i.e., *true* and *false* in Fig. 6.6).

Interaction indicates the invocation of an input/output operation between the system and an actor. *Internal* indicates that the system alters its internal state. *Exit* represents the end of a use case flow, while *Abort* represents the termination of an anomalous execution flow. Fig. 6.7 shows the models generated from the use cases in Table 6.1. For each step identified as *Interaction*, *Include*, *Internal*, *Condition* or *Exit*, a *Step* instance is generated and linked to the previous *Step* instance.

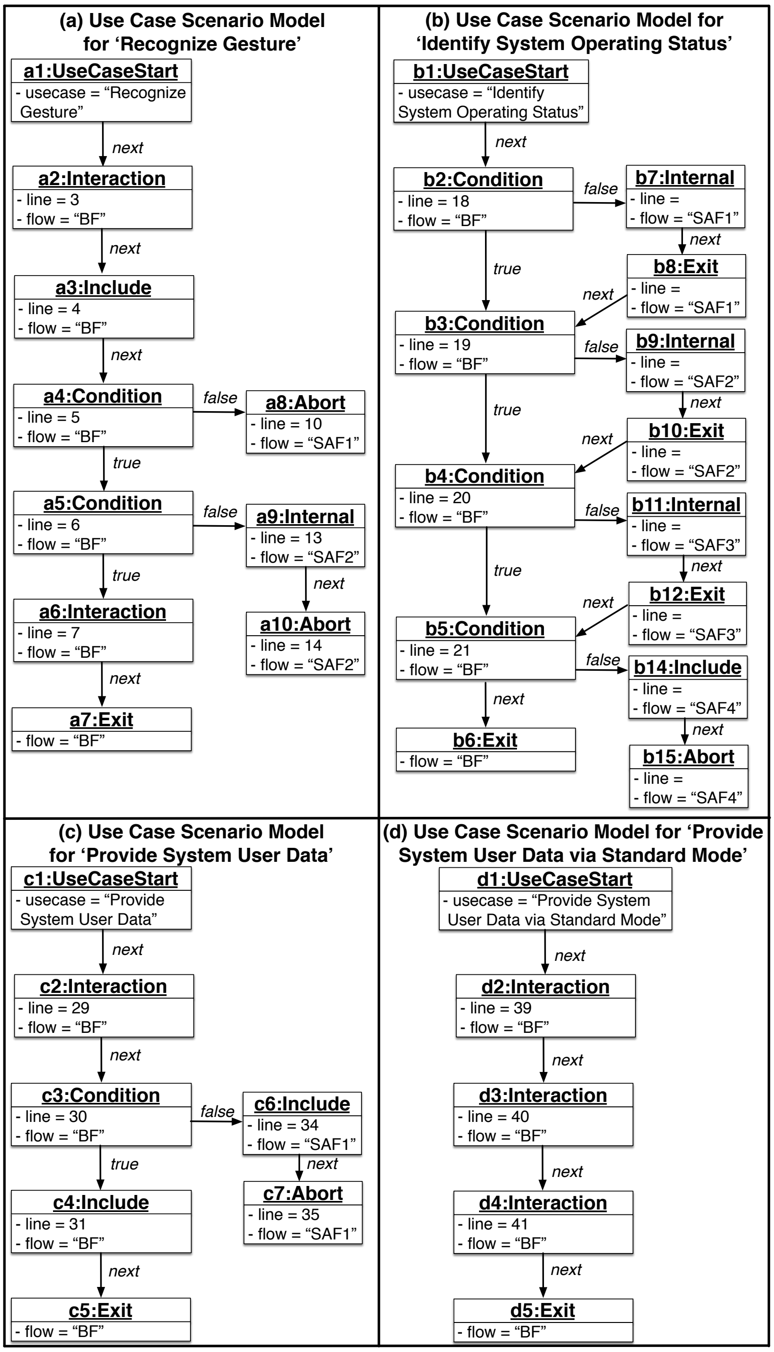


Figure 6.7. Use Case Scenario Models Generated from the Use Case Specifications in Table 6.1

For each alternative flow, a *Condition* instance is created and linked to the *Step* instance of the first step of the alternative flow (e.g., *a4* and *a5* in Fig. 6.7(a)). For multiple alternative flows on the same condition, *Condition* instances are linked to each other in the order they follow in the specification. For alternative flows that return back to the reference flow, an *Exit* instance is linked to the *Step* instance that represents the reference flow step (e.g., *next* between *b8* and *b3* in Fig. 6.7(b)).

For alternative flows that abort, an *Abort* instance is created and linked to the *Step* instance of the previous step (e.g., *a8*, *a10*, *b15* and *c7* in Fig. 6.7). For the end of the basic flow, there is always an

Exit instance (e.g., *a7*, *b6*, *c5* and *d5* in Fig. 6.7).

6.5.2.2 Identification of Tested Use Case Scenarios

We automatically identify tested use case scenarios in a use case specification. A scenario is a sequence of steps that begins with a *UseCaseStart* instance and ends with an *Exit* instance in the use case model. Each use case scenario captures a set of interactions that should be exercised during the execution of a test case.

The function *identifyTestedScenarios* (see Line 12 in Fig. 8) implements a depth-first traversal of use case models to identify tested scenarios. It visits alternative flows which are tested together with previously visited alternative flows by the same test case.

Fig. 6.8 shows three tested scenarios extracted from the scenario models in Fig. 6.7(a) and (c). The scenario in Fig. 6.8(a) executes the true branch of the *Condition* instance *a5* in Fig. 6.7(a), while the scenario in Fig. 6.8(b) executes the false branch of the same *Condition* instance. The scenario in Fig. 6.8(c) executes the basic flows in Fig. 6.7(c) and Fig. 6.7(d).

6.5.2.3 Identification of Test Cases for Use Case Scenarios

We use trace links between test cases and use case specifications to retrieve test cases for a given scenario. The accuracy of test case retrieval depends on the granularity of trace links. Companies may follow various traceability strategies [Ramesh and Jarke, 2001], and generate links in a broad range of granularity (e.g., to use cases, to use case flows or to use case steps). We implement a traceability metamodel which enables the user to generate trace links at different levels of granularity (see Fig. 6.9(a)).

Fig. 6.9 (b) gives part of the traceability model for trace links, assigned by engineers, between two test cases and the use cases *Recognize Gesture* and *Identify System Operating Status* in Table 6.1. Test case *t1* is traced to the basic flows of *Recognize Gesture* and *Identify System Operating Status* (i.e., ($t1 \xrightarrow{tl1} f1$) and ($t1 \xrightarrow{tl3} f3$)), while test case *t2* is traced to the specific alternative flow *SAF2* of *Recognize Gesture* (i.e., ($t2 \xrightarrow{tl2} f2$)).

We retrieve, using the trace links in Fig. 6.9(b), *t1* for the scenario in Fig. 6.8(a) since it is the only scenario executing the basic flows of *Recognize Gesture* and *Identify System Operating Status*. The scenario in Fig. 6.8(b) executes the alternative flow *SAF2* of *Recognize Gesture* (see *a9* and *a10* in Fig. 6.8(b)). Therefore, we retrieve *t2* for the scenario in Fig. 6.8(b).

In the context of our case study, as it is often the case, trace links are assigned from test cases to only basic and alternative flows without indicating the execution order of the flows. There are cases where we need finer-grained trace links to retrieve test cases.

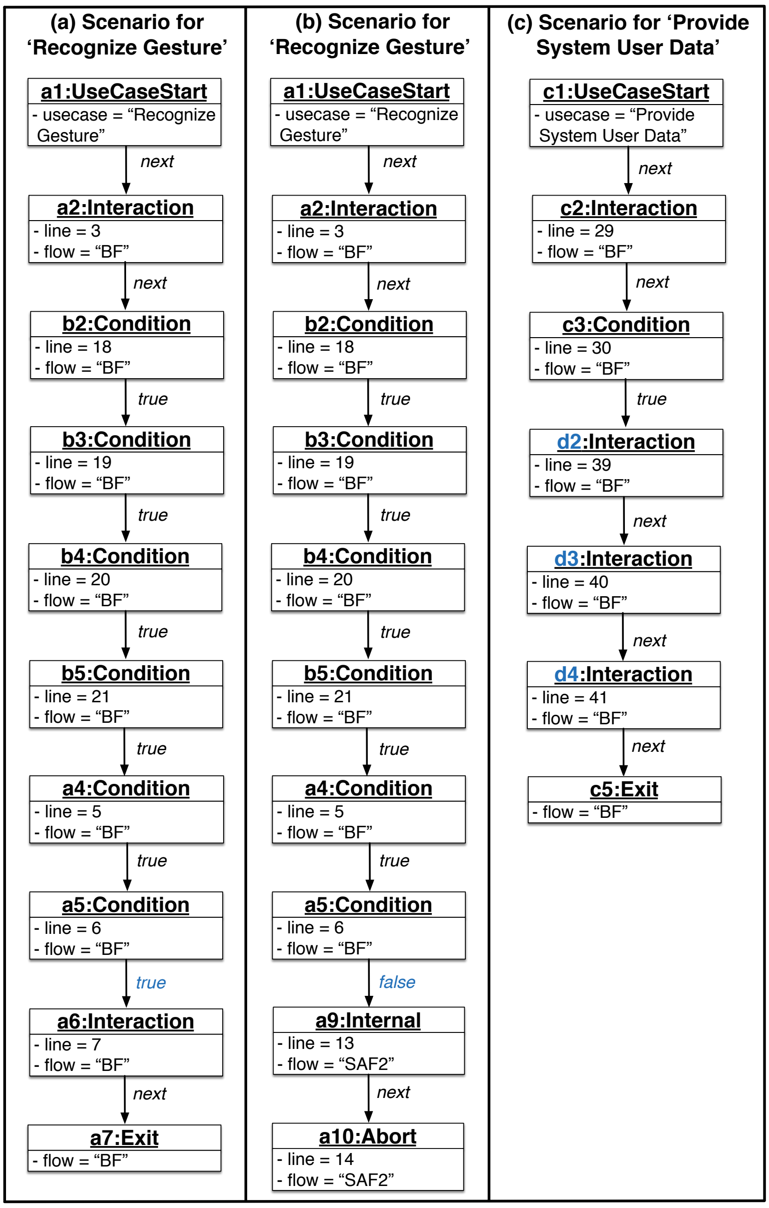


Figure 6.8. Some Tested Use Case Scenarios

First, we need finer-grained trace links when multiple scenarios take the same alternative flows with different orders. In such a case, our approach needs the execution order of alternative flows to match test cases and scenarios (see the attribute *order* in Fig. 6.9(a)).

Second, we need finer-grained trace links when there are more than one scenario taking the same bounded or global alternative flow. Those alternative flows refer to more than one step in a reference flow. Hence, a scenario can take a bounded/global alternative flow from different reference flow steps; we need trace links indicating the reference flow step in which the flow is taken (see the "*to*" from *TraceLink* to *Step* in Fig. 6.9(a)). If we do not have trace links at the right level of granularity, we ask the user to match scenarios and test cases.

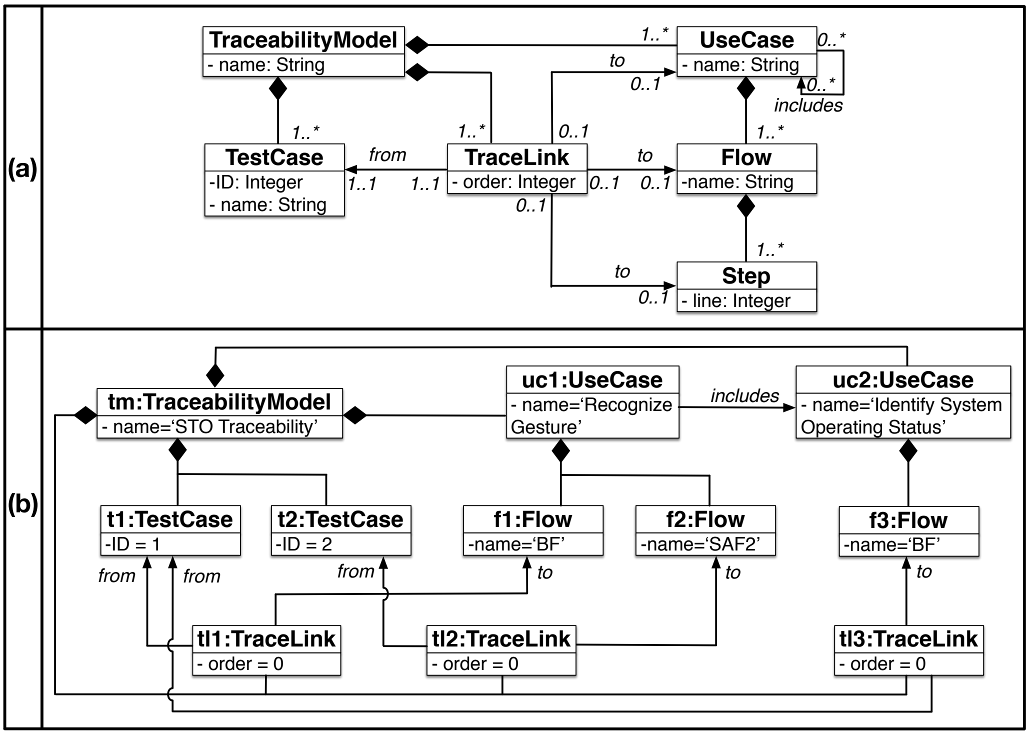


Figure 6.9. (a)Traceability Metamodel and (b) Example Model

6.5.2.4 Impact Identification

We analyze the impact of configuration changes on use case scenarios to classify retrieved test cases as *obsolete*, *retestable* and *reusable*. To this end, we devise an algorithm (see Alg. 9) which takes as input a use case scenario to be analyzed (s_{old}), a set of test cases verifying the scenario (T), a use case specification for the new product (u), and a triple of the sets of configuration changes (dc) produced in Step 2. If there is no change impacting the scenario, test cases verifying the scenario are classified as *reusable* (Line 8 in Alg. 9). For any change in the scenario (e.g., removing a use case step), the test cases are classified as either *retestable* or *obsolete* (Line 5) as shown in Table 6.3, which describes how test cases are classified based on the types of changes affecting the variant elements covered by a scenario. For instance, if a configuration change adds a condition step to a scenario in which the condition refers to an input entity, the test case verifying the scenario is classified as *obsolete* because the new condition requires a change in the inputs of the test case (see rule R4 in Table 6.3).

For the example configuration changes identified in Section 6.5.1, the scenarios in Fig. 6.8(a) and (b) are classified as *retestable* while the scenario in Fig. 6.8(c) is classified as *obsolete*. The tuple $\langle B18, B19\rangle$ represents an updated decision; the unselected optional bounded alternative flow of the use case *Recognize Gesture* is selected in the new product (see Section 6.5.1). The selected optional flow contains a condition, i.e., *"voltage fluctuation is detected"* in Line 10 in Table 3.2, which does not refer to any entity in the input steps. Since the condition step is added in the scenarios in Fig. 6.8(a) and (b), these two scenarios are classified as *retestable*. The triples $\langle B11, B12, B14\rangle$, $\langle B11, B12, B15\rangle$, $\langle B11, B12, B16\rangle$, and $\langle B11, B12, B17\rangle$ in Fig. 6.5 represent updated decisions for the use case *Provide*

System User Data (see Section 6.5.1). Some of the unselected output steps are selected while one selected output step is unselected and the order of the output steps are updated in the basic flow of *Provide System User Data* (see Fig. 6.5). Therefore, the test case verifying the scenario in Fig. 6.8(c) for the basic flow of *Provide System User Data* is classified as *obsolete* (see rules R6, R7 and R9 in Table 6.3).

Alg. 9: Algorithm for analyzeImpact

```
Inputs : Old use case scenario $s_{old}$,
         Set of test cases $T$,
         New use case specification $u$,
         Decision changes $dc$
Output: Quadruple of sets of classified test cases classified
         (OBSOLETE, REUSE, RETEST, NEW)
1  model ← generateScenarioModel(u)
2  Let inst be the UseCaseStart instance in model
3  Let $s_{new}$ be an empty scenario
4  if (there is at least one change in dc for $s_{old}$) then
5      classified ← analyzeChangesOnScenario($s_{old}$, dc, T)
6      NEW ← identifyNewScenarios(model, $s_{old}$, $s_{new}$, inst)
7  else
8      REUSE ← T
9  end if
10 returnclassified
```

We process scenarios impacted by the configuration changes to identify new scenarios for the new product (Line 6 in Alg. 9). Furthermore, for each new scenario, we provide guidance to support the engineers in the implementation of test case(s). To this end, we devise an algorithm (Alg. 10) which takes as input a use case model of the new product (*sm*), a use case step in the model (*inst*), a use case scenario of the previous product (s_{old}) that has been exercised by either an obsolete or retestable test case, and a new scenario (s_{new}) which is initially empty. The algorithm generates a set of triples $\langle\, s_{new}, s_{old}, G \,\rangle$, where s_{new} is the new scenario identified, s_{old} is the old scenario of the previous product, and G is the guidance, i.e., a list of suggestions indicating how to modify test cases covering s_{old} to generate test cases covering s_{new}.

In Fig. 10, the algorithm follows a depth-first traversal of *sm* by following use case steps in *sm* that have corresponding steps in s_{old}. To this end, when traversing condition steps, the algorithm follows alternative flows taken in s_{old} (Lines 8-11). More precisely, whenever a *Condition* instance is encountered, the algorithm checks if the *Condition* instance exists also in s_{old} (Line 8). If so, the algorithm proceeds with the condition branch taken in s_{old}, i.e., the step following the *Condition* instance in s_{old} (Line 11); otherwise, it takes the condition branch(es) which have not yet been taken

Table 6.3. Changes in Use Case Scenarios and Classification of System Test Cases

Rule ID	Change in the Use Case Scenario	Test Case Classification	Rationale
R1	Add or remove an internal step	Retestable	Internal use case steps represent internal system operations (e.g., reset of counters) and do not directly affect system-actor interactions. Therefore, a test case does not need to be modified to exercise a scenario including added or deleted internal steps (e.g., a new internal step does not imply an additional test input or an update in the test oracle). The test case can be executed against the new product without any change; however, the system may not behave as expected (e.g., because of a faulty implementation of a new internal use case step) and thus the test case is classified as retestable.
R2	Update the order of an internal step	Retestable	Since internal use case steps do not directly affect system-actor interactions, a test case does not need to be modified in the presence of a change in the order of internal steps (i.e., a different sequence of internal steps does not imply an update in test inputs or oracles). However, the system may not behave as expected (e.g., because of a faulty implementation of the new order of an internal step) and thus the test case is classified as retestable.
R3	Add or remove a condition step where the condition refers exclusively to state variables	Retestable	Condition steps are used to verify properties of input entities and/or state variables. A condition step, in practice, restricts the execution of a use case scenario to a subset of the values assigned to the input entities and/or state variables verified by the condition. State variables are used to model the system state, while input entities describe system inputs provided by actors. The addition and removal of condition steps that verify the properties of state variables reflect changes in the internal behaviour of the system but not in the system-actor interactions. Therefore, a test case is not modified in the presence of added/removed condition steps that only verify the properties of state variables (e.g., such a new condition step does not imply an update in test inputs and oracle). However, the system may not behave as expected (e.g., because of a faulty implementation of the changed state variables) and thus the test case is classified as retestable.
R4	Add or remove a condition step where the condition refers to an input entity	Obsolete	Adding or removing a condition step referring to input entities may imply an update in the test inputs if the test input values do not satisfy the changed condition. Since we do not inspect executable test cases in our analysis, it is not possible to determine if the test cases of the previous product already provide the values that fulfill the changed condition. To be conservative, we consider test cases of scenarios impacted by such changes as obsolete thus forcing engineers to verify if the test input values exercise the scenario.
R5	Update the order of a condition step	Obsolete	When old and new scenarios differ regarding the order in which condition steps appear, then the behaviour triggered by the test case of the previous product might not be the same in the new product (e.g., if the steps that define the variables verified by the condition are between the condition steps that have been changed). Therefore, we consider a test case that exercises an old scenario affected by such changes as obsolete.
R6	Add or remove an input/output step	Obsolete	Input and output use case steps represent system-actor interactions. Therefore, the implementation of the test case needs to be modified to exercise the targeted scenario when input and output steps are added or removed (e.g., a new input step implies an additional test input in the test case).
R7	Update the order of an input/output step	Obsolete	Since input and output use case steps represent system-actor interactions, the implementation of the test case needs to be modified to exercise the targeted scenario when the order of input and output steps is updated (e.g., a new order of input steps implies an update in the sequence of test inputs).
R8	Remove an alternative flow	Obsolete	Alternative flows capture sequences of interactions taking place under certain execution conditions. If a use case scenario of the previous product covers an alternative flow that does not exist in the new product, the corresponding test case should be considered as obsolete because the interactions verified by the test case cannot take place with the new product.
R9	Multiple changes in the use case scenario	Obsolete or Retestable	A test case is classified as *obsolete* if there is at least one change in the scenario that makes the test case obsolete. A test case is classified as *retestable* if there are no changes in the scenario that make the test case obsolete and if there is at least one change in the scenario that makes the test case retestable.

in s_{new} (Lines 12-30).

Alternative flows may lead to execution loops; this happens when alternative flows resume the execution of steps belonging to the originating flows. In our current implementation we generate scenarios that cover each loop body once. To this end, when processing condition steps, the algorithm checks if the branches that may lead to cycles have already been traversed (i.e., Lines 14 and 22). If it is the case, the traversal of the scenario is directed towards the branch that brings the scenario out of the cycles (i.e, the true branch for specific alternative flows and the false branch for bounded or global alternative flows as shown in Lines 15 and 23, respectively).

The generation of s_{new} terminates when an *Exit* or *Abort* step is reached (Line 32). The only exception is that of *Exit* steps of included use cases, which lead to another step that follows the *Include* step (Line 33). Before adding s_{new} to the result tuple, we automatically compare s_{old} and s_{new}, and determine their differences to generate guidance for new test cases (G in Line 37). We provide a set of suggestions for adding, removing and updating test case steps corresponding to added, removed and updated use case steps in s_{old} and s_{new}.

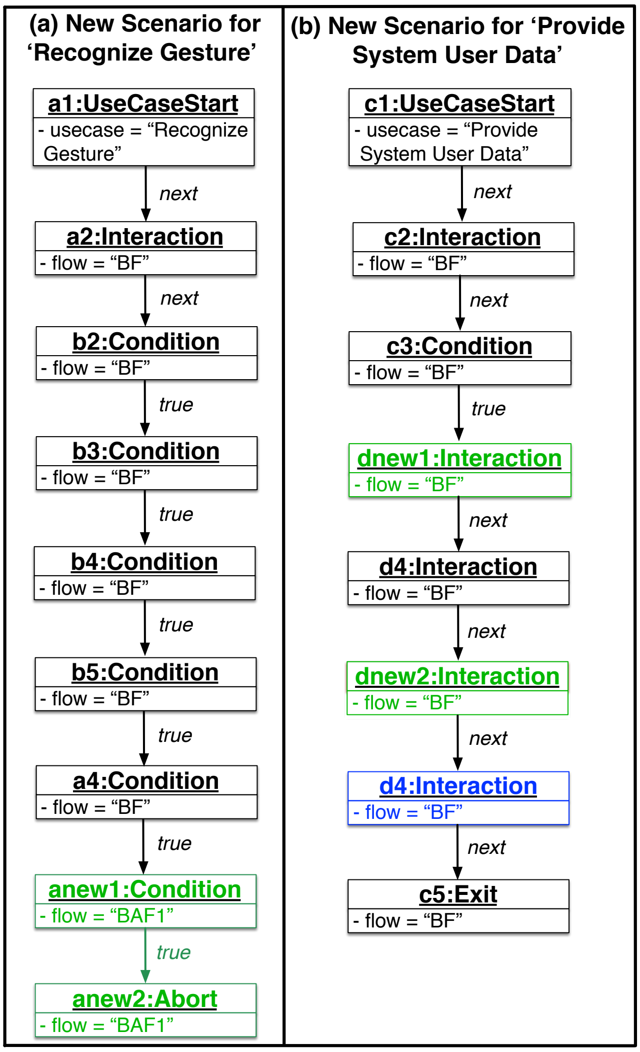


Figure 6.10. Two New Scenarios Derived from the Scenarios in Fig. 6.8

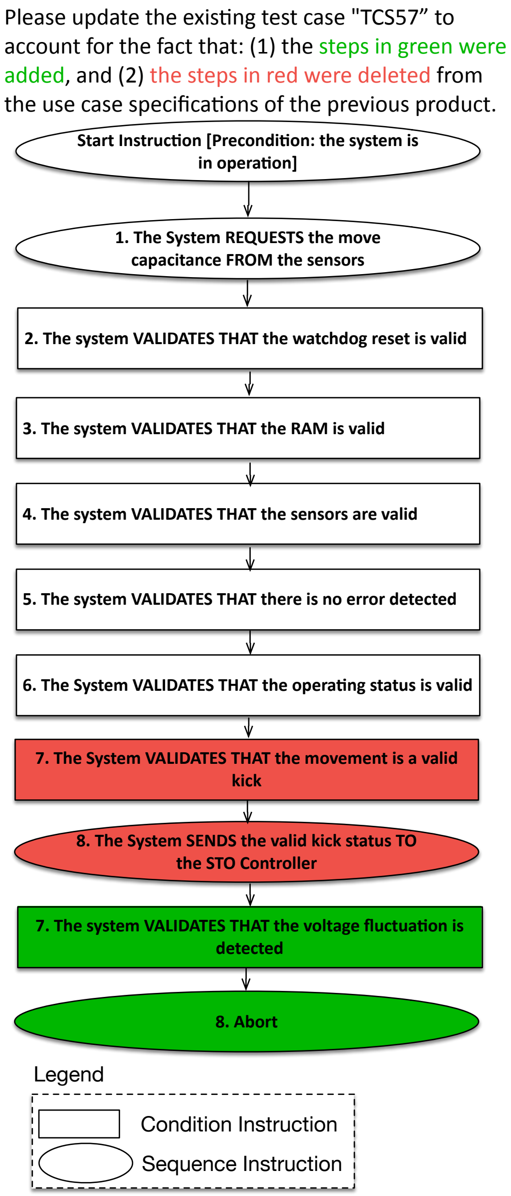
Fig. 6.10 gives two new scenarios derived from the scenarios in Fig. 6.8. Fig. 6.10(a) is derived from Fig. 6.8(a) and (b); Fig. 6.10(b) is derived from Fig. 6.8(c). The new scenario in Fig. 6.10(a) executes the new selected optional bounded alternative flow in which the use case *Recognize Gesture* aborts due to the voltage fluctuation (see Lines 8-12 in Table 3.2). While traversing *sm* for s_{old} in Fig. 6.8(a), the new *Condition* instance *anew1* and the new *Abort* instance *anew2* (green-colored in

Fig. 6.10(a)) are added in s_{new} to execute the bounded alternative flow. s_{new} in Fig. 6.10(b) executes the basic flow of the use case *Provide System User Data* of the new product where the order of one step is updated (blue-colored in Fig. 6.10(b)) and some new steps are introduced (green-colored) while some others are removed.

Alg. 10: Algorithm for identifyNewScenarios

```
Inputs : New scenario model $sm$, model instance $inst$, old scenario $s_{old}$, new scenario $s_{new}$
Output: Set of triples of new scenario, old scenario and guidance $S$
Let $S$ be the empty set for triples of old scenario, new scenario and guidance
if (inst is a UseCaseStart, Interaction or Internal instance) then
    addToScenario($inst$, $s_{new}$)
    $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $inst.next$)
end if
if (inst is a Condition instance) then
    addToScenario($inst$, $s_{new}$)
    if (inst exist in $s_{old}$) then
        Let $t$ be the instance after $inst$ in the branch taken in $s_{old}$
        Let $t_{new}$ be the instance corresponding to $t$ in $sm$
        $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $t_{new}$)
    else
        if (inst represents a condition in a specific alternative flow) then
            if (inst and inst.false exist together in $s_{new}$) then
                $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $inst.true$)
            else
                $s_{cpy} \leftarrow$ clone($s_{new}$)
                $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $inst.true$)
                $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{cpy}$, $inst.false$)
            end if
        else
            if (inst and inst.true exist together in $s_{new}$) then
                $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $inst.false$)
            else
                $s_{cpy} \leftarrow$ clone($s_{new}$)
                $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $inst.false$)
                $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{cpy}$, $inst.true$)
            end if
        end if
    end if
end if
if (inst is an Exit or Abort instance) then
    if (inst is an Exit instance for the included use case) then
        $S \leftarrow S \cup$ identifyNewScenarios($sm$, $s_{old}$, $s_{new}$, $inst.next$)
    else
        addToScenario($inst$, $s_{new}$)
        $G \leftarrow$ generateGuidance($s_{old}$, $s_{new}$)
        $S \leftarrow S \cup \{< s_{new}, s_{old}, G >\}$
    end if
end if
return $S$
```

Fig. 6.11 shows the generated guideline to modify the test case verifying the retestable scenario in Fig. 6.8(a) for the new scenario in Fig. 6.10(a). We use red and green colors, with a legend, on the scenario to explain the impacted parts of the corresponding test case. The red steps are deleted while the green ones are added to the scenario. Using this information, the engineer adds and deletes test case steps to cover the new scenario.



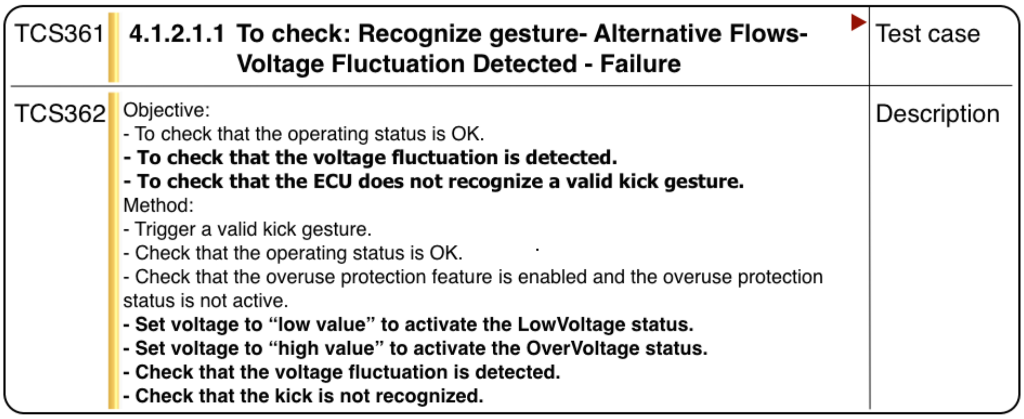
Please update the existing test case "TCS57" to account for the fact that: (1) the steps in green were added, and (2) the steps in red were deleted from the use case specifications of the previous product.

Start Instruction [Precondition: the system is in operation]
1. The System REQUESTS the move capacitance FROM the sensors
2. The system VALIDATES THAT the watchdog reset is valid
3. The system VALIDATES THAT the RAM is valid
4. The system VALIDATES THAT the sensors are valid
5. The system VALIDATES THAT there is no error detected
6. The System VALIDATES THAT the operating status is valid
7. The System VALIDATES THAT the movement is a valid kick
8. The System SENDS the valid kick status TO the STO Controller
7. The system VALIDATES THAT the voltage fluctuation is detected
8. Abort
Legend
Condition Instruction
Sequence Instruction

Figure 6.11. PUMConf's User Interface for Guidance

Fig. 6.12 shows the header of the test case verifying the new scenario in Fig. 6.10(a) with the description of the functions under test. For simplification, we omit the implementations of the executable test case. We use the guidance to derive the new test case from the test case in Fig. 6.2(a) verifying the scenario in Fig. 6.8(a). The bold lines in Fig. 6.12 are the new objectives and methods of the test case that correspond to the new use case steps in Fig. 6.10(a) (i.e., *anew1* and *anew2*).

A new scenario might be derived separately from multiple old scenarios. After all the new scenarios are identified for the new product, we automatically detect such new scenarios and provide



TCS361	**4.1.2.1.1 To check: Recognize gesture- Alternative Flows- Voltage Fluctuation Detected - Failure**	Test case
TCS362	Objective: - To check that the operating status is OK. **- To check that the voltage fluctuation is detected.** **- To check that the ECU does not recognize a valid kick gesture.** Method: - Trigger a valid kick gesture. - Check that the operating status is OK. - Check that the overuse protection feature is enabled and the overuse protection status is not active. **- Set voltage to "low value" to activate the LowVoltage status.** **- Set voltage to "high value" to activate the OverVoltage status.** **- Check that the voltage fluctuation is detected.** **- Check that the kick is not recognized.**	Description

Figure 6.12. System Test Case derived from the System Test Case in Fig. 6.2(a) of the Scenario in Fig. 6.8(a)

guidance for only the test cases of the old scenarios from which the engineer generates the new test cases with the least possible changes (Line 19 in Alg. 8). We rank those old scenarios according to the number of changes. If the number of changes are the same, we give priority to scenarios with more changes removing test case steps. We assume that removing test case steps is more convenient than adding new steps. For instance, our approach derives the new scenario in Fig. 6.10(a) from two scenarios in Fig. 6.8(a) and (b). To generate a test case verifying the new scenario in Fig. 6.10(a), the engineer can modify one of the test cases verifying the scenarios in Fig. 6.8(a) and (b). In Fig. 6.10(a), our approach provides guidance for both scenarios because the number of changes and the number of removed and added test case steps are the same for the two scenarios.

6.5.3 Step 4: Impact Report Generation

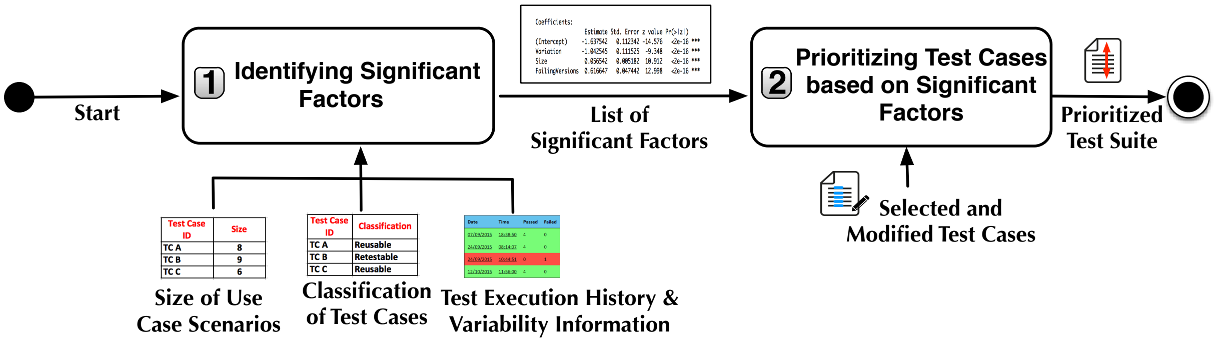
We automatically generate an impact report from the classified test cases of each previous product in a product line (Step 4 in Fig. 6.4). To enable engineers to select test cases from more than one test suite and thus maximize the number of test cases that can be inherited from previous products, we compare all the test suites in the product line and identify sets of new scenarios and reusable and retestable test cases for the product line. Assume that there are N previous products in a product line. S_{new1}, S_{new2}, ..., and S_{newN} are the sets of new scenarios. T_{reus1}, T_{reus2}, ..., and T_{reusN} are the sets of reusable test cases; T_{ret1}, T_{ret2}, ..., and T_{retN} are the sets of retestable test cases we identify when we compare a new product with each previous product. To minimize the number of new test cases the engineer needs to generate, we compute the intersection of the sets of new scenarios ($S_{new} = S_{new1} \cap S_{new2} \cap ... \cap S_{newN}$). The scenarios which are not in the intersection of the sets are covered by at least one reusable or retestable test case in one of the previous products. Therefore, we take the union of the sets of reusable test cases ($T_{reus} = T_{reus1} \cup T_{reus2} \cup ... \cup T_{reusN}$) and the union of the sets of retestable test cases ($T_{ret} = T_{ret1} \cup T_{ret2} \cup ... \cup T_{retN}$). If a test case is considered both retestable and reusable (i.e., $(T_{reus} \cap T_{ret}) \neq \emptyset$), we list the previous products in which the test case is identified as retestable and reusable. The engineer decides the classification of the test case based on the test suite of the previous product he chooses for the new product.

Based on the system under test, engineers decide whether to select test cases from a single test suite or from multiple test suites in the product line. For example, if multiple products include different setup procedures (e.g., due to different HW architecture or library versions being used) that need to be executed at the beginning of each test case, it is more practical to select test cases from a single test suite.

6.6 Prioritization of System Test Cases in a Product Family

Test case prioritization is implemented as a pipeline (see Fig. 6.13). The pipeline takes as input the test suite of the new product, the test execution history of the previous products (i.e., the outcome of each test case of the product test suite, for each previous product and version), the size of the use case scenarios exercised by the test cases, the classification of the test cases (i.e., reusable or retestable), and the variability information of the product line. Based on a prediction model using these factors, the test cases of the given test suite are sorted to maximize the likelihood of executing failing test cases first.

The prioritization pipeline gives the highest priority to test cases covering new scenarios (i.e., scenarios not available for previous products) since they exercise features that have never been tested before. The prioritization of retestable and reusable test cases is instead driven by a set of factors typically correlated with the presence of faults, according to the relevant literature (e.g., [Srikanth et al., 2005] [Engström et al., 2011] [Wong et al., 1997] [Rothermel et al., 2001] [Li et al., 2007]): *the number of previous products in which the test case failed*, *the number of previous products' versions in which the test case failed*, *the size of the scenario exercised by the test case*, *the degree of variability in the use case scenario exercised by the test case*, and *the classification of the test case (i.e., reusable or retestable)*. Note that different versions of a product share the same test suite because functional requirements do not vary across the versions of the same product. The number of previous products in which the test case failed and the number of versions in which the test case failed capture the fault proneness of the test cases, a factor typically considered by other test case prioritization approaches [Srikanth et al., 2005] [Engström et al., 2011]. The size of the use case scenario exercised by a test case is measured in terms of the number of use case steps it contains. The scenario size captures the complexity of the operations performed by the system during the execution of the test case, under the assumption that longer scenarios require more complex software implementations. Implementation complexity is one of the factors considered in other requirements-based prioritization approaches [Srikanth et al., 2005]. The degree of variability in the use case scenario exercised by a test case is measured by counting the number of decision elements included in the use case scenario. In the presence of high variability, it is more likely that some of the system properties verified by the test case are not implemented properly. Finally, the classification of a test case as retestable is considered for prioritization since, by definition, the scenario exercised by a retestable test case might be affected by changes in behaviour and thus may trigger a failure.


Start
1 Identifying Significant Factors
List of Significant Factors
2 Prioritizing Test Cases based on Significant Factors
Prioritized Test Suite
Selected and Modified Test Cases
Size of Use Case Scenarios
Classification of Test Cases
Test Execution History & Variability Information

```
Coefficients:
                 Estimate Std. Error z value Pr(>|z|)
(Intercept)     -1.637542   0.112342 -14.576   <2e-16 ***
Variation       -1.042545   0.111525  -9.348   <2e-16 ***
Size             0.056542   0.005182  10.912   <2e-16 ***
FailingVersions  0.616647   0.047442  12.998   <2e-16 ***
```

Test Case ID	Size
TC A	8
TC B	9
TC C	6

Test Case ID	Classification
TC A	Reusable
TC B	Retestable
TC C	Reusable

Date	Time	Passed	Failed
07/09/2015	18:38:50	4	0
24/09/2015	08:14:07	4	0
24/09/2015	10:44:51	0	1
12/10/2015	11:56:00	4	0

Figure 6.13. Overview of the Test Case Prioritization Pipeline

All these factors mentioned above may have varying importance for test case prioritization in different product lines due to technical and organizational factors. Some factors may even not significantly affect test case prioritization for some product lines. To account for the changing importance of risk factors on test case prioritization, the pipeline first identifies the factors significantly correlated with the presence of failures and prioritizes test cases based on a prediction model relying on such factors.

The prioritization pipeline includes two steps. In Step 1, *Identifying significant factors*, our approach automatically identifies significant factors for prioritizing the test cases of a new product. To this end, we employ logistic regression [Jr. et al., 2013], i.e., a predictive analysis to determine the relationship between one dependent binary variable (i.e., the failure of a test case) and one or more independent variables, which might be either numeric (e.g., the number of the products in which the test case failed in the past) or binary (e.g., the fact that a test case has been classified as retestable).

In our context, the logistic regression model estimates the logarithm of the odds that a test case fails. The logistic regression model is trained using variability information, the size of the use case scenarios exercised by the test cases, the classification of the test cases, and the execution history of the test cases for previous products. The logistic regression model has the following form:

$$ln\left(\frac{p(TC_x)}{1-p(TC_x)}\right) = \beta_0 + \beta_1 * V + \beta_2 * S + \beta_3 * FP + \beta_4 * FV + \beta_5 * R$$

where $p(TC_x)$ is the probability that test case TC_x fails, V is the degree of variability of the scenario exercised by the test case (i.e., the number of decision elements in the scenario), S is the size of the use case scenario exercised by the test case (i.e., the number of steps), FP is the number of failing products, FV is the number of failing versions, and R indicates whether the test case has been classified as retestable. β_0 is the intercept, while $\beta_1 ... \beta_5$ are coefficients which are derived, using the iteratively reweighted least squares approach [Coleman et al., 1980], to estimate the effect size on the failure probability.

We rely on the R environment [Rpr, 2018] to derive the logistic regression model. Our toolset

Table 6.4. Excerpt of the Training Data Set used for Logistic Regression

Product ID	Version ID	Test Case ID	Fails	Retestable	Size of the Use Case Scenario	Degree of Variability of the Scenario	# of Previous Products in which it Fails	# of Previous Versions in which it Fails
P1	V1	TC1	1	0	8	2	0	0
P1	V1	TC2	0	0	4	1	0	0
P1	V2	TC1	1	0	8	2	0	1
P1	V2	TC2	0	0	4	1	0	0
P1	V3	TC1	0	0	8	2	0	2
P1	V3	TC2	0	0	4	1	0	0
P1	V4	TC1	0	0	8	2	0	2
P1	V4	TC2	0	0	4	1	0	0
P2	V1	TC1	1	1	9	3	1	2
P2	V1	TC2	0	0	4	1	0	0
P2	V1	TC3	0	0	4	1	0	0
P2	V2	TC1	0	1	9	3	1	3
P2	V2	TC2	1	0	4	1	0	0
P2	V2	TC3	0	0	4	1	0	0
P2	V3	TC1	0	1	9	3	1	3
P2	V3	TC2	0	0	4	1	0	1
P2	V3	TC3	0	0	4	1	0	0
P3	V1	TC1	1	1	9	3	2	3
P3	V1	TC2	1	1	5	2	1	1
P3	V1	TC3	0	0	4	1	0	0
P3	V2	TC1	1	1	9	3	2	4
P3	V2	TC2	0	1	5	2	1	2
P3	V2	TC3	0	0	4	1	0	0

automatically generates from the available data the training data set to be processed by the R environment. Table 6.4 shows an excerpt of an example training data set generated by our toolset.

Table 6.4 includes the failure history of products $P1$, $P2$ and $P3$ to be used to prioritize the test cases for $P4$. Each row in Table 6.4 reports the information belonging to a single test case executed against a version of a product. The first and second columns represent the product and its version, respectively. The third column reports the test case identifier, while the fourth column indicates whether the test case fails (i.e., the dependent variable). The rest of the columns in Table 6.4 represent independent variables used to predict failure. The fifth column indicates if the test case has been classified as retestable. The sixth column reports the size of the use case scenario exercised by the test case (i.e., the number of steps). The seventh column reports the degree of variability of the scenario exercised by the test case (i.e., the number of decision elements in the scenario). Table 6.4, for instance, shows that test case $TC1$ executed against $P2$ covers nine use case steps while the same test case covers eight use case steps when executed against $P1$; this is due to the covered use case scenario in $P2$ including one additional variant element than the use case scenario covered in $P1$ (see column *Degree of Variability*). Test case $TC3$ has been introduced in $P2$ to cover one additional use case scenario not present in $P1$. The eighth and ninth columns report the number of products and the number of versions in which the test case fails, respectively.

To identify the significant factors for test case prioritization, we apply *the p-value method of hypothesis testing* based on Wald test [Rice, 2007]. The method relies on the failure probability predicted by the regression model to determine whether there is evidence to reject the null hypothesis that *there is no relationship between the two variables*. The p-value indicates the likelihood of observing the data points when the null hypothesis is true. Therefore, if the p-value is smaller than a given threshold (we use 0.05) then it is unlikely that the dataset has been generated by chance and, consequently, the null hypothesis can be rejected (i.e., there is a relationship between the factor and the dependent variable). In the model, we keep the given factors whose p-value is smaller than the threshold. To automatically determine significant factors, we rely on the p-value computed by the Wald test on the logistic regression model trained by including all the factors. Finally, we derive a new, multivariate logistic regression model that includes only the significant factors. For example, the logistic regression model derived for one of the products used in our empirical evaluation (see P4 in Section 6.8) is the following:

$$ln\left(\frac{p(TC_x)}{1-p(TC_x)}\right) = -1.50 - 0.25 * V + 0.04 * S + 0.53 * FV - 1.01 * R$$

This model, for example, does not include the number of failing products (*FP*) since it is not significant according to the computed p-value.

The generated logistic regression model is a predictive model that returns based on the significant factors the probability that a test case fails. In Step 2, *Prioritize test cases*, we prioritize test cases by relying on the probability calculated by the regression model. The test cases are sorted in descending order of probability and presented to the engineer.

6.7 Tool Support

We have implemented our test case selection and prioritization approach as an extension of PUMConf. Section 6.7.1 provides the layered architecture of the tool while we describe the tool features in Section 6.7.2. For accessing the tool, see: `https://sites.google.com/site/pumconf/`.

6.7.1 Tool Architecture

Fig. 6.14 shows the layered architecture of our tool PUMConf. It is composed of three layers: (i) the *User Interface (UI) layer*, (ii) the *Application layer*, and (iii) the *Data layer*.

We briefly introduce each layer and explain the new components, i.e., the gray boxes in Fig. 6.14.

User Interface (UI) Layer. This layer supports creating and viewing PL and PS use case models (i.e., use case diagrams and specifications) and system test cases, and displaying the generated impact reports. We employ Papyrus (`https://www.eclipse.org/papyrus/`) for use case diagrams and IBM Doors (`www.ibm.com/software/products/ca/en/ratidoor/`) for use case specifications and system test cases. The impact reports are visualized as part of IBM Doors output

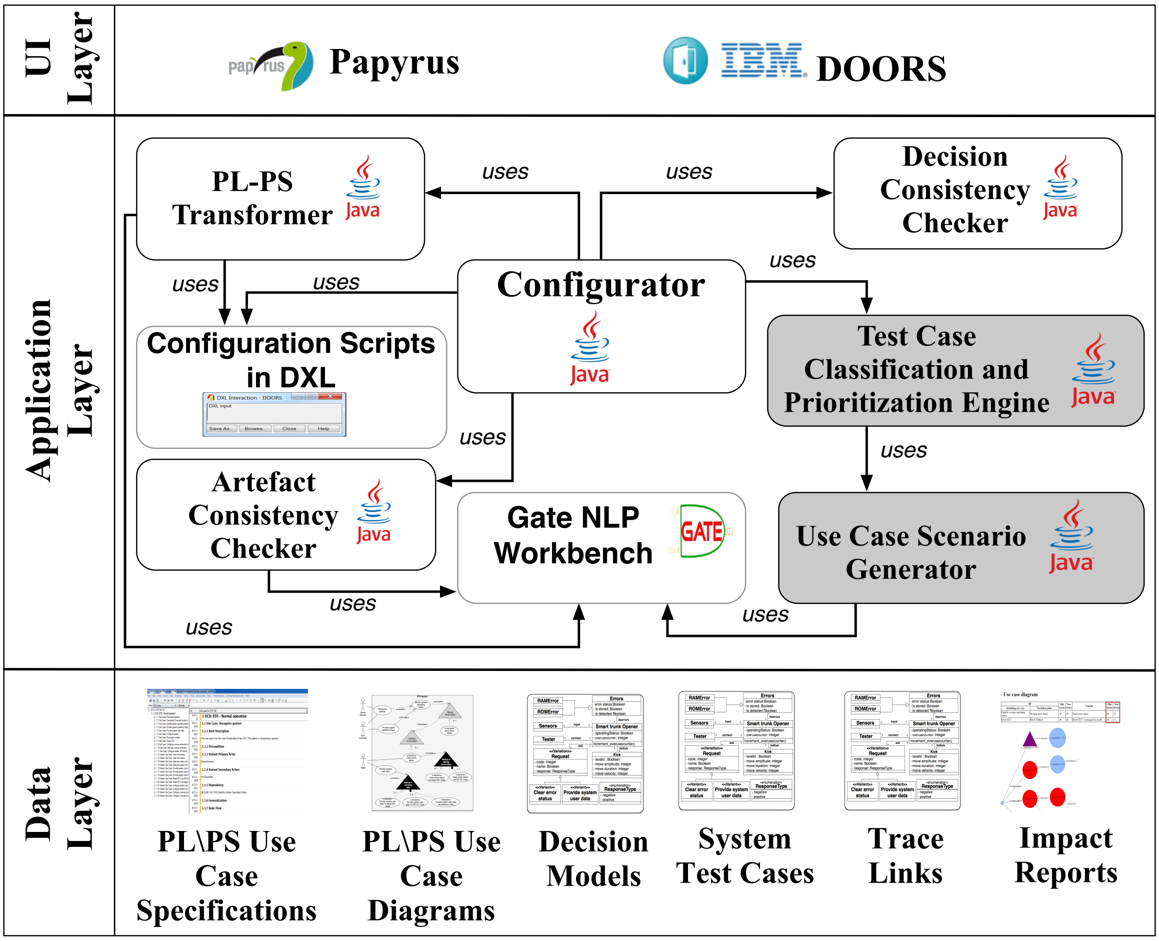


Figure 6.14. Layered Architecture of PUMConf

using JGraph (`https://www.jgraph.com/`), Microsoft Excel (`https://products.office.com/en/excel/`) and html.

Application Layer. With the new components, this layer supports the main activities of our proposed approach in Fig. 6.3: *classifying system test cases*, *providing guidance to modify system test cases*, and *prioritizing the selected system test cases*.

The *Configurator* component coordinates the other components in the application layer. The *Artifact Consistency Checker* and *Decision Consistency Checker* components were introduced in Chapter 5. We further implemented some new components: *Test Case Classification and Prioritization Engine* and *Use Case Scenario Generator*. The *Use Case Scenario Generator* component also employs the GATE workbench to extract control flow information, i.e., the order of alternative flows and their conditions, from use case specifications. With the extracted control flow information, it identifies the new and already tested use case scenarios to be used by the *Test Case Classification and Prioritization Engine* component to classify the system test cases for the new product in a product line and to provide guidance to modify the system test cases for the new use case scenarios that have not been tested before. To prioritize system test cases, the *Test Case Classification and Prioritization Engine* employs R scripts (`https://www.rdocumentation.org/`) to implement logistic regression.

Data Layer. The PL and PS use case specifications are stored in the native IBM DOORS format while the PL and PS use case diagrams are stored as UML models. The decision models are saved in Ecore [Ecl, 2018]. We generate the impact reports as Microsoft Excel spreadsheets and html pages. Depending on industrial practice, the trace links between use case specifications and system test cases can be saved in Excel spreadsheets or in IBM DOORS link database.

6.7.2 Tool Features

We describe the main features of our tool: *classifying system test cases for the new product*, *providing guidance to modify test cases*, and *prioritizing the selected system test cases for the new product*.

Classifying system test cases. This feature supports Step 2, *Classify System Test Cases for the New Product*, in Fig. 6.3. Before the analyst decides which system test cases to be run for the new product, the tool automatically classifies the system test cases of the previous product(s) for the new product.

Providing guidance to modify system test cases. As part of Step 2, *Classify System Test Cases for the New Product*, in Fig. 6.3, the tool automatically provides guidance to modify some obsolete and retestable test cases to cover new, untested use case scenarios in the new product.

Fig. 6.11 shows the generated guidelines to modify the test case verifying the retestable use case scenario in Fig. 6.8(a) to cover the new use case scenario in Fig. 6.10(a). We use red and green colors, with a legend, on the use case scenario to explain the impacted parts of the corresponding system test case. The red steps in the use case scenario are deleted while the green ones are added to the use case scenario. Using this information, the test engineer adds and deletes the corresponding test case steps in the system test case to cover the new use case scenario.

Prioritizing the selected system test cases. This feature supports Step 4, *Prioritize System Test Cases for the New Product*, in Fig. 6.3. Once the test engineer selects the system test cases of the previous product(s) and generates the new system test cases of the new use case scenarios for the new product, the tool automatically prioritizes the selected and generated system test cases using multiple risk factors given in the test execution history.

6.8 Evaluation

Our objective is to assess, in an industrial context, whether our approach could improve test case reuse and reduce testing effort. This empirical evaluation aims to answer the following research questions:

- *RQ1. Does the proposed approach provide correct test case classification results?* This research question aims to evaluate the precision and recall of the procedure adopted to classify the test cases developed for previous products.

- *RQ2. Does the proposed approach accurately identify new scenarios that are relevant for testing a new product?* This research question aims to evaluate the precision and recall of the approach in identifying the new scenarios to be tested for a new product (i.e., new requirements not covered by existing test cases).
- *RQ3. Does the proposed approach successfully prioritize test cases?* This research question aims to determine whether the approach is able to effectively prioritize system test cases that trigger failures and thus can help minimize testing effort while retaining maximum fault detection power.
- *RQ4. Can the proposed approach significantly reduce testing costs compared to current industrial practice?* This research question aims to determine to what extent the proposed approach can help significantly reduce the cost of defining and executing system test cases.

6.8.1 Subject of the Study

The subject of our study is the Smart Trunk Opener (STO) system developed by our industry partner IEE. STO has been selected for the assessment of our approach since it is a relatively new project at IEE involving multiple customers requiring varying features. The development history of the STO product line includes five products delivered to different car manufacturers. STO customers include major car manufacturers working in the European, Asian and US markets, with 2017 sales ranging from 200,000 to 3 million vehicles. For each product, IEE engineers developed multiple versions, each sharing the same functional requirements but differing with respect to non-functional requirements (e.g., hardware selection or performance optimizations). In total, STO includes 54 versions.

Table 6.5. Overview of the STO Product Line Use Cases

	# of Use Cases	# of Variation Points	# of Basic Flows	# of Alternative Flows	# of Steps	# of Optional Alternative Flows	# of Optional Steps
Essential UCs	15	5	15	70	269	5	14
Variant UCs	14	3	14	132	479	8	13
Total	29	8	29	202	748	13	27

To develop the STO system, IEE engineers elicited requirements as use cases from an initial customer. For each new customer, they cloned the current use cases and identified differences to produce new use cases. Table 6.5 provides an overview of the STO product line. The data in Table 6.5 shows that the system implements 29 use cases, each one being fairly complex since the use cases in total include 202 alternative flows (i.e., alternative cases to be considered when implementing the use case). The STO product line is highly configurable, with 14 variant use cases, 8 variation points, 13 optional alternative flows and 27 optional steps. STO has the size and characteristics of typical embedded product line systems managing automotive components. To apply the proposed approach, we have considered STO requirements written according to PUM [Hajri et al., 2015] [Hajri et al., 2018b]. Table 6.6 reports information about the STO products including the number of versions for each product. In Table 6.6, the products are sorted according to their delivery date, with P1 being the first product of the product line, and P5 being the last.

Table 6.6. Details of the Configured Products in the STO Product Line

Product ID	# of Versions	# of Use Case Elements			# of Test Cases
		Use Cases	Use Case Flows	Use Case Steps	
P1	22	28	236	689	110
P2	8	25	169	568	86
P3	10	28	234	685	96
P4	5	26	212	618	83
P5	9	28	238	695	113

The different STO products are characterized by different test suites of different sizes while the same test suite is shared by all the versions of the same product since their functional requirements do not vary. The test cases have been traced to the use case specifications by IEE engineers. Column *# Test Cases* in Table 6.6 shows, for every STO product, the number of test cases belonging to the functional test suite of the product.

6.8.2 Experiment Setup

Our approach for test case classification can be applied using single-product settings (i.e., to classify and prioritize test cases that belong to a previous product) and whole-line settings (i.e., to classify test cases of multiple previous products). To evaluate our approach for test case classification and to spot differences in terms of classification results with the two configurations (e.g., number of test cases that can be reused), we applied the approach using both settings. To evaluate test case prioritization, we prioritized test suites developed to test different STO products. We applied test case prioritization to the entire test suite since its execution is required by safety standards for every product being released.

6.8.3 Results

6.8.3.1 RQ1

To answer RQ1, we, together with IEE engineers, inspected the classification results produced by the approach. We evaluated the approach in terms of the average precision and recall we computed over the three different classes according to standard formulas [Sokolova and Lapalme, 2009]. In our context, a true positive is a test case correctly classified according to the expected class (e.g., a

Table 6.7. Test Case Classification Results for Single-Product Settings

Classified Test Suite	Product to be Tested	# of Reusable	# of Retestable	# of Obsolete	Precision	Recall
P1	P2	94	2	14	1.0	1.0
P1	P3	105	2	3	1.0	1.0
P1	P4	102	2	6	1.0	1.0
P1	P5	84	22	4	1.0	1.0
P2	P3	85	0	1	1.0	1.0
P2	P4	83	0	3	1.0	1.0
P2	P5	67	16	3	1.0	1.0
P3	P4	91	0	5	1.0	1.0
P3	P5	77	17	2	1.0	1.0
P4	P5	77	5	1	1.0	1.0

reusable test case classified as reusable). A false positive is a test case incorrectly classified as being part of a given class (e.g., a retestable test case classified as reusable). A false negative is a test case that belongs to a given class but has not been classified as such (e.g., a reusable test case not classified as reusable).

Tables 6.7 and 6.8 provide the results for the single-product and whole-line settings, respectively. The first two columns report the ID of the product(s) whose test suite(s) have been considered for classification and the ID of the product being tested, respectively. The next three columns provide the number of test cases belonging to the three classes. The last two columns indicate precision and recall. We observe that the approach has perfect precision and recall. This is the result of meticulous requirements modeling practices in place at IEE where functional requirements are documented by means of use cases together with proper traceability to test cases. These practices enable a precise identification of impacted scenarios and consequently the correct classification of test cases. It is typical for companies, like IEE, developing embedded, safety-critical systems, since requirements need to be traced and tested to comply with international safety standards (e.g., ISO 26262 [ISO, 2018]). To apply the approach, we relied on the trace links assigned by IEE engineers to use cases and system test cases. We did not need to ask IEE engineers to provide additional trace links to match scenarios with test cases since all the trace links were at the right level of granularity required by our approach.

6.8.3.2 RQ2

To answer RQ2, we checked if the new scenarios are exercised by the test cases in the manually implemented test suites of the new products. If so, we consider those new scenarios relevant. In addition, we, together with IEE engineers, checked whether the new scenarios not exercised by the test suites of the new products are relevant for testing these new products. We classified the new

Table 6.8. Test Case Classification Results for Whole-Line Settings

Classified Test Suites	Product to be Tested	Reusable	Retestable	Obsolete	Precision	Recall
P1	P2	94	2	14	1.0	1.0
P1, P2	P3	107	0	2	1.0	1.0
P1, P2, P3	P4	102	0	12	1.0	1.0
P1, P2, P3, P4	P5	93	15	1	1.0	1.0

scenarios as true positive (i.e., a scenario identified by our approach and relevant for testing), false positive (i.e., a scenario identified by our approach but not relevant for testing), and false negative (i.e., a scenario tested by IEE but not identified by our approach). We computed precision and recall accordingly.

Tables 6.9 and 6.10 report the results obtained using the single-product and whole-line settings, respectively. The third, fourth, fifth columns provide the number of relevant scenarios identified by our approach, and, among these, the number of scenarios tested and not tested by IEE engineers. The sixth column (*Not Relevant*) indicates the number of irrelevant identified scenarios. The columns named *New Scenarios Not Identified* provide the number of scenarios tested by IEE engineers but not identified by our approach. The last two columns report precision and recall. All the new scenarios identified by our approach are relevant; they are covered by the test cases produced by IEE engineers. Consequently, the approach has perfect precision and recall.

In addition, we observe from Table 6.10 that the availability of additional products in the whole-line settings enables the identification of additional new scenarios, and consequently more accurate testing. This is what happens for product P5, in which the whole-line settings lead to the identification of 14 new scenarios. Five of these new scenarios have not been tested by engineers in any of the existing products. More precisely, the test suites of P1 and P3 enable the identification of four and three scenarios not tested in the test suite of P5, respectively; only two of these scenarios are tested by both for a total of five new scenarios identified. This difference between existing test suites is explained by the fact that certain test teams have defined more complete test suites (i.e., the test team for P1 and P3). Since new scenarios are identified based on obsolete test cases, for products with more complete test suites, the availability of more test cases may lead to the identification of additional new scenarios.

6.8.3.3 RQ3

To answer RQ3, we applied our test case prioritization approach to sort the test cases in the test suites of four STO products (i.e., P2, P3, P4 and P5). In total, we built four regression models, one for each STO product. To evaluate the quality of the prediction, we relied on historical data. Using test execution history, we verified that higher priority was given to test cases that have failed. Because of

Table 6.9. Relevance of Scenarios Identified using Single-Product Settings

Classified Test Suite	Product to be Tested	New Scenarios Identified				New Scenarios Not Identified (FN)	Precision	Recall
		Relevant (TP)	Tested by Engineers	Not Tested	Not Relevant (FP)			
P1	P2	3	3	0	0	0	1.0	1.0
P1	P3	3	3	0	0	0	1.0	1.0
P1	P4	2	1	1	0	0	1.0	1.0
P1	P5	27	23	4	0	0	1.0	1.0
P2	P3	1	1	0	0	0	1.0	1.0
P2	P4	1	1	0	0	0	1.0	1.0
P2	P5	22	22	0	0	0	1.0	1.0
P3	P4	1	1	0	0	0	1.0	1.0
P3	P5	26	23	3	0	0	1.0	1.0
P4	P5	10	10	0	0	0	1.0	1.0

Table 6.10. Relevance of Scenarios Identified using Whole-Line Settings

Classified Test Suites	Product to be Tested	New Scenarios Identified				New Scenarios Not Identified (FN)	Precision	Recall
		Relevant (TP)	Tested by Engineers	Not Tested	Not Relevant (FP)			
P1	P2	3	3	0	0	0	1.0	1.0
P1, P2	P3	1	1	0	0	0	1.0	1.0
P1, P2, P3	P4	0	0	0	0	0	1.0	1.0
P1, P2, P3, P4	P5	14	9	5	0	0	1.0	1.0

this, we prioritized the test cases that belong to the test suite originally developed by IEE engineers and ignored the new test scenarios we have identified (Section 6.5). This did not introduce bias in the evaluation since test cases exercising new scenarios are always on top of the prioritized test suite and their execution is always necessary independently from their predicted capability to discover faults. In the following, we discuss our results including the identification of significant factors and the effectiveness of the prioritized test suites.

Table 6.11 provides detailed information about the significant factors identified. Column *Significant factors* lists the significant factors identified for each product. When more historical information is available, more factors significantly correlate with the presence of faults. For example, we observe that the classification of a test case as retestable becomes significant after three products are included in the development history of the product line. This can be explained by the fact that updated configuration decisions impact a limited number of scenarios (i.e., the number of retestable test cases is

Table 6.11. Analysis of Significant Factors identified by Logisitic Regression

Classified test suites	Product to be tested	Significant factors	Odds Ratio
P1	P2	V; S; FV	0.35; 1.08; 2.09
P1, P2	P3	V; S; FV	0.35; 1.06; 1.85
P1, P2, P3	P4	V; S; FV; R	0.78; 1.04; 1.71; 0.36
P1, P2, P3, P4	P5	V; S; FP; FV; R	0.36; 1.04; 0.92; 1.87; 0.51

Legend: V=Variation, S=Size, FP=Failing Products, FV=Failing Versions, R=Retestable.

Table 6.12. Test Case Prioritization Results

Classified test suites	Product to be tested	AUC ratio (Observed/Ideal)	%Test cases executed to identify		%Failures detected with 50% of the test cases
			All the failures	80% of the failures	
P1	P2	0.98 (65.46/66.48)	72.09%	38.37%	97.43%
P1, P2	P3	0.99 (82/82.48)	41.66%	22.91%	100%
P1, P2, P3	P4	0.97 (71.02/72.97)	51.80%	22.89%	95%
P1, P2, P3, P4	P5	0.95 (101.32/105.97)	26.54%	18.58%	100%

usually low) and thus, this factor only becomes significant when enough examples of retestable test cases have occurred in previous products. As expected, the number of failing products also becomes significant after a sufficient number of products in the product line.

Column *Odds Ratio* presents the odds ratio of each significant factor. The odds ratio captures the effect size of the factor on the outcome of the regression model (i.e., the probability of observing a failure). A value above one indicates that the factor positively contributes to the outcome; a factor below one indicates that the factor negatively contributes to the outcome (note that this may be caused by a statistical interaction with another factor). The results show that the number of failing versions is the factor that impacts most positively the probability of failure. It is highly likely that a test case that failed in the past will fail again, which is in line with previous research results. The odds ratio for the number of failing versions varies between 1.71 and 2.09. We observe that, expectedly, the number of failing products statistically interacts with the number of failing versions. This has been determined by running logistic regression on each factor separately. Certain factors show a positive regression parameters when considered alone but become negative when interacting with other factors in the regression model. In this case, this is probably due to the two factors being correlated.

To evaluate the effectiveness of test case prioritization, we measured the percentage of test cases to be executed to trigger all the failures, and compared our approach with the ideal case that executes all the failing test cases first. Table 6.12 summarizes our findings. For all the products in our evaluation, our approach identifies more than 80% of the failures by executing less than 50% of the test cases (see

Table 6.13. Test Development Costs Savings

Product to be tested	Test Cases To Be Implemented using the Proposed Approach	
	Single-Product Settings	Whole-Line Settings
P2	3/99 (3%)	3/99 (3%)
P3	1/86 (1%)	1/108 (1%)
P4	1/92 (1%)	0/102 (0%)
P5	10/92 (11%)	14/122 (11%)

Table 6.14. Development Process Savings

Product to be tested	Test Suite Size	Number of Failures	Test cases to be executed to identify all the failures	
			Current Practice	Proposed Approach
P2	86	39	84 (97.67%)	62 (72.09%)
P3	96	27	80 (83.33%)	40 (41.66%)
P4	83	20	77 (92.77%)	43 (51.80%)
P5	113	14	69 (61.06%)	30 (26.54%)

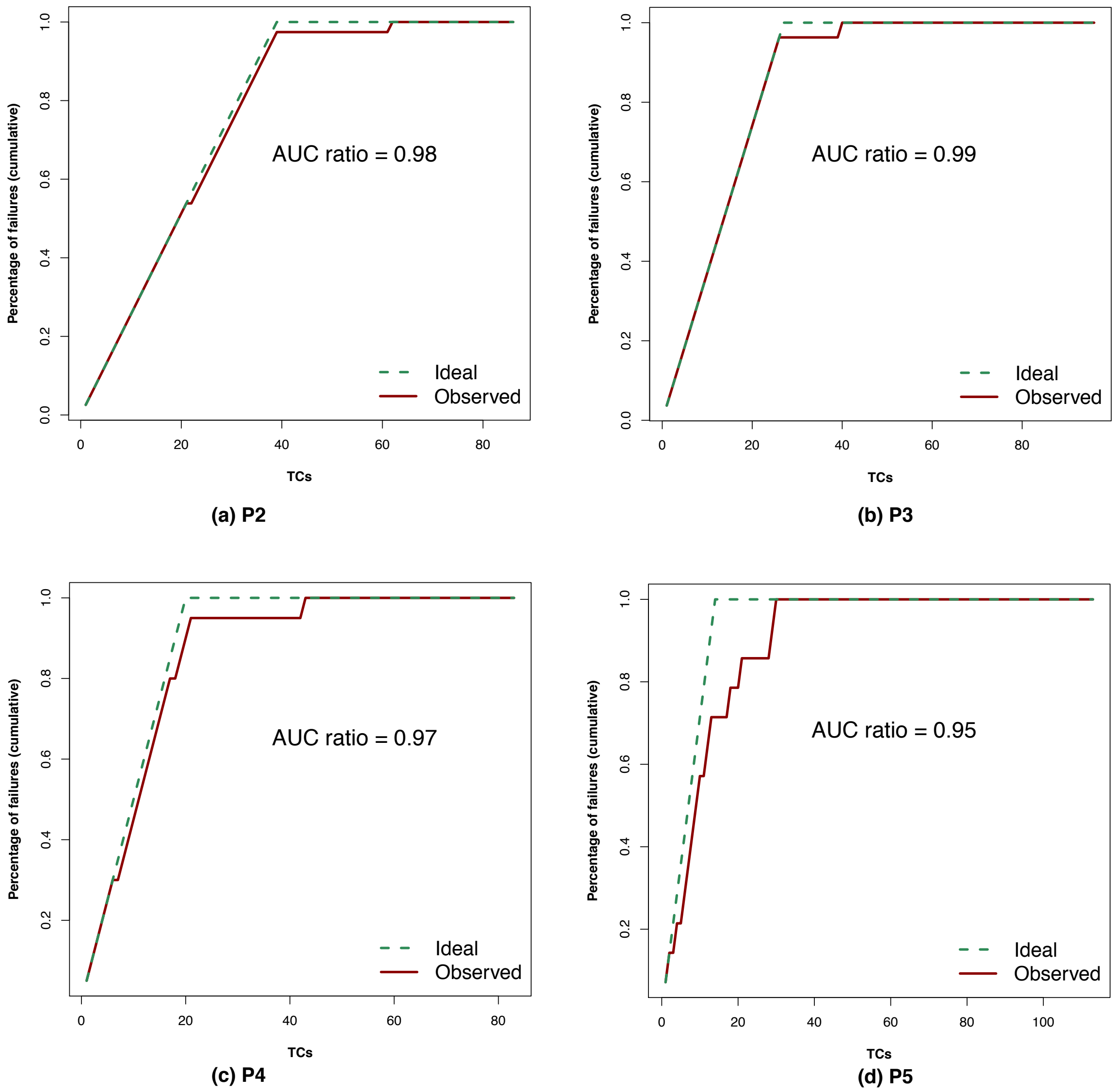
Columns *%Failures detected with 50% of the test cases* and *%Test cases executed to identify 80% of the failures*). We notice that the number of test cases required to trigger all the failures drops below 60% when the test execution history of at least two products becomes available (see Column *%TCs executed to identify all the failures*). In the case of P5, for example, the execution of 27% of the test cases is sufficient to trigger all failures. This is explained by the fact that newer products are more mature (i.e., they tend to fail less frequently) but is also due to logistic regression models improving over time. Indeed, for newer products, though it is less probable to identify failing test cases (they are fewer), our approach remains accurate at giving higher priority to failing test cases. This capability is particularly relevant for industry since the early identification of failures enables early maintenance activities and, consequently, speeds up the product release.

To compare our approach with the ideal case, we computed the area under curve for the cumulative percentage of failures triggered by executed test cases for both the ideal case and prioritization, and computed the AUC ratio of the two. Fig. 6.15 shows the two curves. The best result is achieved when the AUC ratio is equal to one (i.e., the AUC for the observed data matches the ideal AUC). The results show that the proposed approach achieves impressive results since the AUC ratio is always greater than or equal to 0.95.

6.8.3.4 RQ4

Our test case classification and prioritization approach may reduce both (i) test case development costs (i.e., the number of test cases that need to be designed and implemented by engineers to test the software) and (ii) software development time (e.g., by detecting more failures at early stages of testing).

As a surrogate metric to measure the savings, for each product of the STO product line, we report



(a) P2

(b) P3

(c) P4

(d) P5

Figure 6.15. Prioritization results: percentage of failures detected after running *x* prioritized test cases.

the number and percentage of test cases that can be reused when adopting the proposed approach (see Table 6.13). Columns *Single-Product Settings* and *Whole-Line Settings* report the results achieved by the approach when reusing only the test cases inherited from one previous product and from the test suites of all the previous products in the product line, respectively. As seen in these two columns, the effort required to implement test cases is very limited since, with the proposed approach, engineers need to implement only the test cases required to cover new scenarios. For instance, in the whole-line settings for product P4, engineers do not need to implement any test case at all. Instead, testing teams at IEE currently do not rely on approaches that support systematic reuse of test cases, a practice which often leads to re-implementing most of the test cases from scratch. Finally, one benefit provided by the whole-line configuration settings is the identification of additional new scenarios, as discussed in Subsection 6.8.3.2; this is the case of P5 where the whole-line configuration settings lead to the identification of four additional scenarios not identified with the single-product settings.

To evaluate the impact of our approach on software development time, we measured the percentage of test cases that need to be executed in order to identify all the failures in a product (see Table 6.14). Column *Current Practice* in Table 6.14 reports the percentage of test cases that need to be executed when considering the order followed by IEE engineers, which is based on domain knowledge. Column *Proposed Approach* reports the results for the proposed approach. For all the products, our approach identifies all the failures with less test cases than the current practice. This is particularly true for product P5 where our approach requires the execution of less than half of the test cases prioritized by engineers. By using our approach, IEE can detect and fix failures earlier and thus speed up their software development.

6.8.4 Threats to Validity

Internal validity. To limit threats to internal validity, we considered the test cases developed by IEE engineers and the historical information collected over the years of system development. To avoid bias in the results, we considered the use case specifications written by IEE engineers and simply reformulated them according to PUM [Hajri et al., 2015] [Hajri et al., 2018b].

External validity. To mitigate the threat to generalizability, we considered a software product line that includes nontrivial use cases, with multiple customers and many sources of variability, in an application domain where product lines are the norm. The fact that STO has been installed on cars developed by major car manufacturers all over the world guarantees that the configuration decisions for STO cover a wide spectrum of possible configurations and that the testing process put in place by IEE adheres to the state-of-the-art quality standards. Based on our experience built over the years with various automotive companies, we expect that the type of configuration decisions characterizing the STO product line and the type and number of test cases developed for STO are good representatives for other types of embedded automotive systems.

6.9 Conclusion

This chapter presents an automated test case classification and prioritization approach that supports use case-driven testing in product lines. The approach automatically classifies and prioritizes, for each new product, system test cases of previous product(s) in a product line, and provides guidance in modifying existing system test cases to cover new use case scenarios that have not been tested in the product line before.

We aimed to improve the test case selection and execution process in product lines by informing engineers about the impact of requirements changes on system test cases in a product family and to minimize manual test case selection and execution effort by automatically and incrementally classifying and prioritizing system test cases, that is to only select the unaffacted test cases and modify the affected ones given a requirements change.

Our test case classification and prioritization approach is built on top of our Product line Use case Modeling method in Chapter 3 and our Product line Use case Model Configurator in Chapter 4, and supported by a tool integrated into IBM DOORS. The key characteristics of our tool support are (1) the automated identification of the impact of requirements changes on system test cases to be classified and selected for the new product, (2) the automated identification of new use case scenarios in the new product that have not been tested in the product line, (3) the automated generation of guidance for modifying existing system test cases to cover those new scenarios, and (4) the automated prioritization of the selected system test cases for the new product. We performed a case study in the context of automotive domain, which suggests that our approach is practical and beneficial to classify and prioritize system test cases in industrial product lines and to provide guidance for modifying existing system test cases for new products in industrial settings.

In this chapter, we answered *Research Question 5 (How can a change in a configuration decision be propagated to other decisions in PL use case models and to system test cases? How can we support the analysts in performing changes? How can we reconfigure PS use case and domain models for decision changes? How can we select and prioritize system test cases for such changes?).* With the test case classification and prioritization approach, we addressed the issues about automated identification of change impact on system test cases, automated test case classification, and automated test case prioritization in product lines.

This work is the last step to achieve our long term objective in this thesis, i.e., the support for change impact analysis and regression test selection that help engineers manage changes in requirements and system test cases in a product family. Our approach does not support the evolution of PL use cases. We still need to address and manage changes in variability aspects of PL use cases such as adding a new variation point in the PL use case diagram, and their impact on test cases in the context of test case selection and prioritization (see the future research directions in Chapter 7).

Chapter 7

Conclusion

This chapter summarizes the research contributions of this dissertation and discusses potential areas for future work.

7.1 Introduction

This chapter concludes on the contributions described in this thesis. First, in Section 7.2, we summarize the problems addressed in the thesis: (i) *modeling variability in requirements with additional traceability to feature models*, (ii) *manual, expensive and error prone configuration of product specific requirements*, (iii) *manual and expensive regression testing in product families*. We reflect on the solutions we proposed in Section 7.3, while Section 7.4 outlines further research directions.

7.2 Problems

In this thesis, we have addressed the following problems in supporting change in product lines for use case-driven development and testing:

- **Modeling Variability in Requirements with Additional Traceability to Feature Models.** Within the context of use case-driven development, there is a need to explicitly document requirements variability in the form of use cases and domain models to be communicated to customers and other stakeholders such as independent test teams during product configuration. The most commonly adopted approach in the literature is relating feature models to use case and domain models. It has shortcomings in terms of additional modeling and traceability effort. Requirements variability documented across various use case flows with several trace links from feature models would have to be communicated to customers and used to configure a product, which in practice, we were told by engineers, is very impractical.

- **Manual, Expensive and Error Prone Configuration of Product Specific Requirements.** Use case-driven product configuration requires a high degree of automation to enable analysts to be interactively guided for their configuration decisions in PL use case and domain models. Any inconsistency in PL use case and domain models may cause invalid configurations. Therefore, before the decision-making process, it should be ensured that all artifacts with variability information, including the use case diagram and specifications, as well as the domain model, are consistent. There may be contradicting configuration decisions and hierarchies among decisions during the decision-making process. Changes on configuration decisions may impact prior decisions as well as on subsequent decisions. While making configuration decisions, analysts need to be interactively informed about contradicting decisions, the order of possible decisions, and the impact of decision changes on other decisions. Without interactive guidance and proper tool support, analysts have to manually identify and fix inconsistent PL artifacts, resolve decision contradictions, and change subsequent decisions, which leads to time-consuming, expensive and error prone configuration.

- **Manual and Expensive Regression Testing in Product Families.** In most of the development environments for product lines, product requirements are elicited from the initial customer and documented as a use case diagram and use case specifications. Test engineers generate system test cases from those use case specifications. For each new product in the product family, engineers manually choose and prioritize, from the existing test suite(s) for the initial/previous product(s), test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly in the new product. This form of test reuse is not supported by any structured, automated test case selection and prioritization method. It is fully manual, error-prone and time-consuming, which leads to ad-hoc change management for system test cases in product lines. Therefore, product line modeling and testing techniques are needed to automate the reuse and adaptation of system test cases in the context of use case-driven development of a product family.

7.3 Solutions

In this section, we explain how we have addressed the aforementioned problems. The proposed techniques tackle manual and error prone requirements configuration and regression testing issues.

- **A modeling method for capturing variability information in Product Line (PL) use case and domain models.** We proposed, applied, and assessed the Product line Use case modeling Method (PUM) that supports variability modeling in PL use case diagrams, specifications, and domain models, without making use of feature models, thus minimizing modeling and traceability overhead. PUM adopts existing PL extensions for use case diagrams and domain models [Halmans and Pohl, 2003] [Ziadi and Jezequel, 2006]. For modeling variability in use case specifications, we introduced new product line extensions for the Restricted Use Case Modeling method (RUCM) [Yue et al., 2013], which includes a template and restriction rules to reduce imprecision and incompleteness in use cases. PUM is supported by a tool relying on Natural Language Processing (NLP) to check the consistency of PL use case and domain models.

- **An approach for automated configuration of Product Specific (PS) use case and domain models.** We proposed, applied, and assessed a use case-driven configuration approach that provides, based on our modeling method (PUM), a degree of configuration automation for effective product-line management in use case-driven development. The approach supports four activities. First, the analysts model the variability information explicitly in a PL use case diagram, its use case specifications, and its corresponding domain model. Second, the analyst is guided to make configuration decisions in an appropriate order. Third, the consistency of configuration decisions is ensured by automatically identifying contradicting decisions. Fourth, PS use case and domain models are automatically generated from PL models and configuration decisions. Our approach is supported by a tool, PUMConf (Product line Use case Model Configurator), relying on NLP and integrated with IBM DOORS. The tool automatically checks the consistency of PL use case and domain models, identifies the partial order of decisions to be made, determines contradicting decisions, and generates PS use case and domain models.

- **A change impact analysis approach for evolving configuration decisions in PL use case models.** We proposed, applied, and assessed a change impact analysis approach that supports, based on our use case-driven modeling and configuration techniques, the evolution of configuration decisions. The approach automates the identification of configuration decisions impacted by decision changes in PL use case models. It supports three activities. First, the analyst proposes a change but does not apply it to the corresponding configuration decision. Second, the impact of the proposed change on other configuration decisions for the PL use case diagram are automatically identified. In the PL use case diagram, variant use cases and variation points are connected to each other with some dependencies, i.e., *require*, *conflict* and *include*. In the case of a changed diagram decision contradicting prior and/or subsequent diagram decisions, such as a subsequent decision resulting in selecting variant use cases violating some dependency constraints, we automatically detect and report them. To this end, we developed a change impact analysis algorithm, which traverses the input PL use case diagram and reasons on prior and subsequent decisions. Based on this, the analyst should decide whether the proposed change is to be applied. Third, the PS use case models are incrementally regenerated only for the impacted decisions after the analyst actually makes all the required changes. To do so, we implemented a model differencing pipeline which identifies decision changes to be used in the reconfiguration of PS models. There are two sets of decisions: (i) the set of previously made decisions used to initially generate the PS use case models and (ii) the set of current decisions including decisions changed after the initial generation of the PS models. Our approach compares the two sets to determine for which decisions we need to incrementally regenerate the PS models. We extended our configurator, PUMConf, to support these three activities

- **An approach for automated classification and prioritization of system test cases in a family of products.** We proposed, applied, and assessed an automated regression testing approach that supports, based on our use case-driven modeling and configuration techniques, product line testing for evolving products of a product family in terms of evolving configuration decisions in PL use cases. The approach automates the identification of system test cases impacted by changes in configuration decisions in PL use case models when a new product is configured. After the initial product is tested, subsequent products are tested using regression testing techniques, i.e., test case selection and prioritization based on configuration decision changes between the previous product(s) and the new product to be tested. Therefore, system test cases for a new product are derived by reusing system test cases for previous product(s), and by identifying use case scenarios of the new product that have not been tested so far in the product family. To reuse the existing system test cases, our approach automatically classifies the system test cases of the previous product(s) as *obsolete*, *retestable*, and *reusable*. An *obsolete* test case cannot be executed on the new product as the corresponding use case scenarios are not selected for the new product. A *retestable* test case is still valid but needs to be rerun to determine the possible impact of changes whereas a *reusable* test case is also valid but does not need to be rerun for the new product. To do so, we implemented a model differencing and test case classification pipeline which identifies changes in the decisions made to configure a product (e.g., selecting an optional use case). The pipeline compares the configuration decisions of the previous and new products to classify the decisions as *new*, *deleted* and *updated*, and to identify the impacted parts of the use case models of the previous product(s). By using the trace links from the impacted parts of the use case models to the system test cases, we automatically classify the existing system test cases to be reused for testing the new product. In addition, we automatically identify the use case scenarios of the new product that have not been tested before, and provide information of how to modify existing system test cases to cover these new, untested use case scenarios, i.e., the impact of use case changes on existing system test cases. System test cases are automatically prioritized based on multiple risk factors such as fault proneness of requirements and requirements volatility in the product line. To this end, we compute a prioritization score for each system test case based on these factors building a statistical prediction model based on historical data. To support these activities, we extended PUMConf.

7.4 Future Research Directions

This thesis covered various applications of automated use case-driven configuration to solve the manual and error-prone change management of requirements and test cases in product families. These applications led to open issues that we will investigate in the future.

- **Change Impact Analysis for Evolving Product Line Use Case Models.** Change can occur both in configuration decisions and variability aspects of PL use case models. For the latter (e.g., making a variant use case essential or changing the cardinality constraint of a variation point), the impact on configuration decisions in each configuration in the product line needs to be assessed and reconfiguration should be considered in PS use case models for impacted configurations. To this end, we plan to develop a change impact analysis approach to identify impacted configurations in the product line and parts of PS models that need to be reconfigured when PL use case models evolve.

- **Classification and Prioritization of Test Cases for Evolving Product Line Use Case Models.** When PL use case models evolve in a product family, the change impact on the execution of system test cases need to be assessed for configurations impacted by changes in the PL use case models. For instance, changing the cardinality constraint of a variation point may cause the unselection of a variant use case selected for a product. All the test cases verifying the previously selected use case will be obsolete as a result of changing the cardinality constraint in the PL use case models. We plan to provide an automated regression test selection approach for system test cases in configurations impacted by changes in PL use case models.

- **Change Impact Analysis and Regression Testing for Evolving Product Specific Use Case Models.** After a product is configured for a customer, there might still be changes in the configured PS use case models. The customer may not want to expose these changes at the level of the product line and therefore to other customers. These changes might be updating some use case steps or alternative flows even when they do not contain variations. Therefore, change impact analysis and regression testing approaches need to be developed to manage such changes for PS use case models and system test cases.

- **Automated Generation of Trace Links between System Test Cases and Use Case Specifications.** Some existing test case generation approaches [Wang et al., 2015a] [Wang et al., 2015b] [Yue et al., 2015a] can be adapted to the product line context to generate system test cases and their trace links in the presence of PL requirements. The adoption of these test case generation approaches may not be feasible in all contexts since such approaches require additional modeling efforts such as use case conditions written in OCL. We would like to investigate the recent advances in NLP, such as semantic role labeling [Jurafsky and Martin, 2000], text chunking [Jurafsky and Martin, 2000], and syntactic and semantic similarity measures [Manning et al., 2008] [Rus et al., 2013], for the generation of trace links between system test cases and use case specifications.

- **Tooling.** We have developed tool support, PUMConf, for the approaches and techniques developed within the context of the thesis. PUMConf has been developed as an IBM DOORS Plug-in. PUMConf relies on Papyrus (`https://www.eclipse.org/papyrus/`) for managing use case diagrams, and IBM Doors (`www.ibm.com/software/products/ca/en/ratidoor/`) for managing use case specifications and test cases. The output of PUMConf are visualized as part of IBM Doors output using JGraph (`https://www.jgraph.com/`) and Microsoft Excel (`https://products.office.com/en/excel/`). We plan to make PUMConf an independent tool in which the user can manage use case diagrams and specifications without relying on any external tool.

List of Papers

Published papers included in this dissertation:

1. Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. **“Applying Product Line Use Case Modeling in an Industrial Automotive Embedded System: Lessons Learned and a Refined Approach”.** In *Proceedings of the 18th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS’15)*, Ottawa, Canada, September 30 - October 2, pp. 338-347, 2015.

2. Ines Hajri. **“Supporting Change in Product Lines within the Context of Use Case-driven Development and Testing”.** In *Proceedings of the Doctoral Symposium at the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE’16)*, Seattle, USA, November 13-16, pp. 1082-1084, 2016.

3. Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. **“PUMConf: a Tool to Configure Product Specific Use Case and Domain Models in a Product Line”.** In *Proceedings of the Tool Demo Track at the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE’16)*, Seattle, USA, November 13-16, pp. 1008-1012, 2016.

4. Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. **“Incremental Reconfiguration of Product Specific Use Case Models for Evolving Configuration Decisions”.** In *Proceedings of the 23rd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ’17)*, Essen, Germany, February 27 - March 2, pp. 3-21, 2017.

5. Ines Hajri, Arda Goknil, and Lionel C. Briand. **“A Change Management Approach in Product Lines for Use Case-Driven Development and Testing”.** In *Proceedings of the Tool Demo and Poster Track at the 23rd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ’17)*, Essen, Germany, February 27 - March 2, 2017.

6. Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. **“Configuring Use Case Models in Product Families”.** In *Software & Systems Modeling (SoSyM)*, Springer, vol. 17(3), pp. 939-971, 2018.

7. Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. **“Change Impact Analysis for Evolving Configuration Decisions in Product Line Use Case Models”.** In Journal of Systems and Software, Elsevier, vol. 139, pp. 211-237, 2018.

8. Ines Hajri, Arda Goknil, Fabrizio Pastore, and Lionel C. Briand. **“Automating Test Case Classification and Prioritization for Use Case-Driven Testing in Product Lines”.** Submitted to a TOSEM journal.

Bibliography

[IEE, 1990] (1990). Institute of electrical and electronics engineers: Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). IEEE.

[GRL, 2009] (2009). Goal-oriented requirements language (grl), `https://www.cs.toronto.edu/km/GRL/`.

[SWE, 2014] (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK V3.0)*. IEEE Computer Society Press.

[CMU, 2018] (2018). Carnegie Mellon University Software Engineering Institute, `https://resources.sei.cmu.edu/library/results.cfm?as_q=inmeta:gsataxonomyoutput~Software%20Product%20Lines`.

[Dop, 2018] (2018). DOPLER (Decision Oriented Product Line Engineering for effective Reuse), `http://www.ase.jku.at/dopler/`.

[Ecl, 2018] (2018). Eclipse EMF, `https://eclipse.org/modeling/emf/`.

[IEE, 2018] (2018). IEE (International Electronics & Engineering) S.A., `http://www.iee.lu/`.

[OCL, 2018] (2018). Object Constraint Language (OCL). http://www.omg.org/spec/OCL/.

[Pur, 2018a] (2018a). pure::variants for IBM Rational DOORS, `http://www.pure-systems.com/DOORS.174.0.html`.

[Pur, 2018b] (2018b). pure::variants, `http://www.pure-systems.com/pure_variants.49.0.html`.

[Rpr, 2018] (2018). The R project, `https://www.r-project.org`.

[Acher et al., 2012] Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., and Merle, P. (2012). Feature model differences. In *CAiSE'12*, pages 629–645.

[Al-Hajjaji et al., 2017a] Al-Hajjaji, M., Kruger, J., Schulze, S., Leich, T., and Saake, G. (2017a). Efficient product-line testing using cluster-based product prioritization. In *AST'17*, pages 16–22.

[Al-Hajjaji et al., 2017b] Al-Hajjaji, M., Lity, S., Lachmann, R., Thum, T., Schaefer, I., and Saake, G. (2017b). Delta-oriented product prioritization for similarity-based product-line testing. In *VACE'17*, pages 34–40.

[Al-Hajjaji et al., 2016] Al-Hajjaji, M., Thum, T., Lochau, M., Meinicke, J., and Saake, G. (2016). Effective product-line testing using similarity-based product prioritization. In *Software and System Modeling*.

[Al-Hajjaji et al., 2014] Al-Hajjaji, M., Thum, T., Meinicke, J., Lochau, M., and Saake, G. (2014). Similarity-based prioritization in software product-line testing. In *SPLC'14*, pages 197–206.

[Alférez et al., 2014] Alférez, M., Bonifácio, R., Teixeira, L., Accioly, P., Kulesza, U., Moreira, A., Araújo, J., and Borba, P. (2014). Evaluating scenario-based spl requirements approaches: the case for modularity, stability and expressiveness. *Requirements Engineering Journal*, 19:355–376.

[Alferez et al., 2008] Alferez, M., Kulesza, U., Moreira, A., Araujo, J., and Amaral, V. (2008). Tracing between features and use cases: A model-driven approach. In *VAMOS'08*, pages 81–88.

[Alférez et al., 2009] Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., and Amaral, V. (2009). Multi-view composition language for software product line requirements. In *SLE'09*, pages 103–122.

[Alves et al., 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring product lines. In *GPCE'06*, pages 201–210.

[Alves et al., 2010] Alves, V., Niu, N., Alves, C., and Valença, G. (2010). Requirements engineering for software product lines: A systematic review. *Information and Software Technology*, 52:806–820.

[Amyot, 2003] Amyot, D. (2003). Introduction to the user requirements notation: Learning by example. *Computer Networks*, 42(3):285–301.

[Anton, 1996] Anton, A. (1996). Goal-based requirements analysis. In *ICRE'96*, pages 136–144.

[Arafeen and Do, 2013] Arafeen, M. J. and Do, H. (2013). Test case prioritization using requirements-based clustering. In *ICST'13*, pages 312–321.

[Armour and Miller, 2001] Armour, F. and Miller, G. (2001). *Advanced Use Case Modeling: Software Systems*. Addison-Wesley.

[Arora et al., 2015a] Arora, C., Sabetzadeh, M., Goknil, A., Briand, L. C., and Zimmer, F. (2015a). Change impact analysis for natural language requirements: An nlp approach. In *RE'15*, pages 6–15.

[Arora et al., 2015b] Arora, C., Sabetzadeh, M., Goknil, A., Briand, L. C., and Zimmer, F. (2015b). NARCIA: an automated tool for change impact analysis in natural language requirements. In *ESEC/SIGSOFT FSE'15*, pages 962–965.

[Arrieta et al., 2017] Arrieta, A., Sagardui, G., Etxeberria, L., and Zander, J. (2017). Automatic generation of test system instances for configurable cyber-physical sytems. *Software Quality Journal*, 25(3):1041–1083.

[Arrieta et al., 2016] Arrieta, A., Wang, S., Sagardui, G., and Etxeberria, L. (2016). Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In *GECCO'16*, pages 1053–1060.

[Arrieta et al., 2019] Arrieta, A., Wang, S., Sagardui, G., and Etxeberria, L. (2019). Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software*, 149:1–34.

[Azevedo et al., 2010] Azevedo, S., Machado, R. J., Braganca, A., and Ribeiro, H. (2010). The UML "extend" relationship as support for software variability. In *SPLC'10*, pages 471–475.

[Azevedo et al., 2012] Azevedo, S., Machado, R. J., Braganca, A., and Ribeiro, H. (2012). On the refinement of use case models with variability support. *Innovations Syst Softw Eng*, 8:51–64.

[Baller et al., 2014] Baller, H., Lity, S., Lochau, M., and Schaefer, I. (2014). Multi-objective test suite optimization for incremental product family testing. In *ICST'14*, pages 303–312.

[Basanieri et al., 2002] Basanieri, F., Bertolino, A., and Marchetti, E. (2002). The cow_suite approach to planning and deriving test suites in uml projects. In *UML'02*, pages 383–397.

[Batory, 2005] Batory, D. (2005). Feature models, grammars, and propositional formulas. In *SPLC'05*, pages 7–20.

[Benavides et al., 2005a] Benavides, D., Ruiz-Cortés, A., and Trinidad, P. (2005a). Using constraint programming to reason on feature models. In *SEKE'05*, pages 677–682.

[Benavides et al., 2010] Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.

[Benavides et al., 2005b] Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2005b). Automated reasoning on feature models. In *CAiSE'05*, pages 491–503.

[Bertolino et al., 2006] Bertolino, A., Fantechi, A., Gnesi, S., and Lami, G. (2006). Product line use cases: Scenario-based specification and testing of requirements. In *Software Product Lines*. Springer.

[Bertolino and Gnesi, 2003] Bertolino, A. and Gnesi, S. (2003). PLUTO: a test methodology for product families. In *PFE'03*, pages 181–197.

[Biddle et al., 2002] Biddle, R., Noble, J., and Tempero, E. (2002). Supporting reusable use cases. In *ICSR'02*, pages 210–226.

[Binkley, 1997] Binkley, D. (1997). Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516.

[Blanes et al., 2014] Blanes, D., Gonzalez-Huerta, J., and Insfran, E. (2014). A multimodel approach for specifying the requirements variability on software product lines. In *ISD'14*, pages 329–336.

[Bonifácio and Borba, 2009] Bonifácio, R. and Borba, P. (2009). Modeling scenario variability as croscutting mechanisms. In *AOSD'09*, pages 125–136.

[Bonifácio et al., 2015] Bonifácio, R., Borba, P., Ferraz, C., and Accioly, P. (2015). Empirical assessment of two approaches for specifying software product line use case scenarios. *Software and Systems Modeling*.

[Bonifácio et al., 2008] Bonifácio, R., Borba, P., and Soares, S. (2008). On the benefits of scenario variability as croscutting. In *EA-AOSD'08*, pages 1–6.

[Botterweck and Pleuss, 2014] Botterweck, G. and Pleuss, A. (2014). Evolution of software product lines. In *Evolving Software Systems*. Springer.

[Braganca and Machado, 2007] Braganca, A. and Machado, R. J. (2007). Automating mappings between use case diagrams and feature models for software product lines. In *SPLC'07*, pages 3–12.

[Briand et al., 2009] Briand, L. C., Labiche, Y., and He, S. (2009). Automating regression test selection based on uml designs. *Information and Software Technology*, 51:16–30.

[Buhne et al., 2006] Buhne, S., Halmans, G., Lauenroth, K., and Pohl, K. (2006). Scenario-based application requirements engineering. In *Software Product Lines*. Springer.

[Buhne et al., 2003] Buhne, S., Halmans, G., and Pohl, K. (2003). Modeling dependencies between variation points in use case diagrams. In *REFSQ'03*, pages 59–69.

[Bürdek et al., 2015] Bürdek, J., Kehrer, T., Lochau, M., Reulig, D., Kelter, U., and Schürr, A. (2015). Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, pages 1–47.

[Cabral et al., 2010] Cabral, I., Cohen, M. B., and Rothermel, G. (2010). Improving the testing and testability of software product lines. In *SPLC'10*, pages 241–255.

[Chen, 1976] Chen, P. P.-S. (1976). The entity-relationship specification - toward a unified view of date. *ACM Transactions on Database Systems*, 1(1):9–38.

[Chen et al., 2002] Chen, Y., Probert, R. L., and Sims, D. P. (2002). Specification-based regression test selection with risk analysis. In *CASCON'02*.

[Chung et al., 1996] Chung, L., Nixon, B. A., and Yu, E. (1996). Dealing with change - an approach using non-functional requirements. *Requirements Engineering*, 1(4):238–259.

[Clarke et al., 2010] Clarke, D., Helvensteijn, M., and Schaefer, I. (2010). Abstract delta modeling. In *GPCE'10*, pages 13–22.

[Cleland-Huang et al., 2005] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., and Christina, S. (2005). Goal-centric traceability for managing non-functional requirements. In *ICSE'05*, pages 362–371.

[Clements and Northrop, 2001] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[Cockburn, 2001] Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.

[Coleman et al., 1980] Coleman, D., Holland, P., Kaden, N., Klema, V., and Peters, S. C. (1980). A system of subroutines for iteratively reweighted least squares computations. *ACM Transactions on Mathematical Software*, 6(3):327–336.

[Czarnecki and Antkiewicz, 2005] Czarnecki, K. and Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, pages 422–437.

[Czarnecki et al., 2005] Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., and Pietroszek, K. (2005). fmp and fmp2rsm: Eclipse plug-ins for modeling features using model templates. In *OOPSLA'05*, pages 200–201.

[Czarnecki et al., 2002] Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U. W. (2002). Generative programming for embedded software: an industrial experience report. In *GPCE'02*, pages 156–172.

[Czarnecki and Helsen, 2006] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

[Czarnecki et al., 2004] Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged configuration using feature models. In *SPLC'04*, pages 266–283.

[de Mota Silveira Neto et al., 2011] de Mota Silveira Neto, P. A., do Carmo Machado, I., McGregor, J. D., de Almeida, E. S., and de Lemos Meira, S. R. (2011). A systematic mapping study of software product lines testing. *Information and Software Technology*, 53:407–423.

[DeMarco, 1978] DeMarco, T. (1978). *Structured Analysis and System Specification*. Prentice Hall.

[Devroey et al., 2017] Devroey, X., Perrouin, G., Cordy, M., Samih, H., Legay, A., Schobbens, P.-Y., and Heymans, P. (2017). Statistical prioritization for software product line testing: An experience report. *Software and Systems Modeling*, 16(1):153–171.

[Devroey et al., 2014] Devroey, X., Perrouin, G., Cordy, M., Schobbens, P.-Y., Legay, A., and Heymans, P. (2014). Towards statistical prioritization for software product lines testing. In *VaMoS'14*, pages 1–7.

[Dhungana et al., 2011] Dhungana, D., Grünbacher, P., and Rabiser, R. (2011). The DOPLER metatool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18:77–114.

[Dintzner et al., 2014] Dintzner, N., Kulesza, U., van Deursen, A., and Pinzger, M. (2014). Evaluating feature change impact on multi-product line configurations using partial information. In *ICSR'15*, pages 1–16.

[Do, 2016] Do, H. (2016). Recent advances in regression testing techniques. In *Advances in Computers*, volume 103.

[do Carmo Machado et al., 2014] do Carmo Machado, I., Mcgregor, J. D., Cavalcanti, Y. C., and De Almeida, E. S. (2014). On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199.

[Dukaczewski et al., 2013] Dukaczewski, M., Schaefer, I., Lachmann, R., and Lochau, M. (2013). Requirements-based delta-oriented spl testing. In *PLEASE'13*, pages 49–52.

[Durán et al., 2016] Durán, A., Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. (2016). FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software and Systems Modeling*.

[e Zehra Haidry and Miller, 2013] e Zehra Haidry, S. and Miller, T. (2013). Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering*, 39(2):258–275.

[Elmasri and Navathe, 2015] Elmasri, R. and Navathe, S. B. (2015). *Fundamentals of Database Systems (7th Edition)*. Pearson.

[Engstrom, 2010] Engstrom, E. (2010). Regression test selection and product line system testing. In *ICST'10*, pages 512–515.

[Engström, 2013] Engström, E. (2013). *Supporting Decisions on Regression test Scoping in a Software Product Line Context - from Evidence to Practice*. PhD thesis, Lund University.

[Engstrom and Runeson, 2011] Engstrom, E. and Runeson, P. (2011). Software product line testing - a systematic mapping study. *Information and Software Technology*, 53:2–13.

[Engström et al., 2011] Engström, E., Runeson, P., and Ljung, A. (2011). Improving regression testing transparency and efficiency with history-based prioritization - an industrial case study. In *ICST'11*, pages 367–376.

[Ensan et al., 2011] Ensan, A., Bagheri, E., Asadi, M., Gasevic, D., and Biletskiy, Y. (2011). Goal-oriented test case selection and prioritization for product line feature models. In *ITNG'11*, pages 291–298.

[Eramo et al., 2012] Eramo, R., Malavolta, I., Muccini, H., Pelliccione, P., and Pierantonio, A. (2012). A model-driven approach to automate the propagation of changes among architecture description languages. *Software and System Modeling*, 11:29–53.

[Eriksson et al., 2004] Eriksson, M., Borstler, J., and Asa, A. (2004). Marrying features and use cases for product line requirements modeling of embedded systems. In *SERPS'04*, pages 73–82.

[Eriksson et al., 2005a] Eriksson, M., Borstler, J., and Borg, K. (2005a). The pluss approach - domain modeling with features, use cases and use case realizations. In *SPLC'05*, pages 33–44.

[Eriksson et al., 2009] Eriksson, M., Borstler, J., and Borg, K. (2009). Managing requirements specifications for product lines - an approach and industry case study. *Journal of Systems and Software*, 82:435–447.

[Eriksson et al., 2005b] Eriksson, M., Morast, H., Borstler, J., and Borg, K. (2005b). The pluss toolkit - extending telelogic doors and ibm-rational rose to support product line use case modeling. In *ASE 2005*, pages 300–304.

[Fan and Zhang, 2006] Fan, S. and Zhang, N. (2006). Feature model based on description logics. In *KES'06*, pages 1144–1151.

[Fantechi et al., 2004a] Fantechi, A., Gnesi, S., John, I., Lami, G., and Dorr, J. (2004a). Elicitation of use cases for product lines. In *PFE'03*, pages 152–167.

[Fantechi et al., 2004b] Fantechi, A., Gnesi, S., Lami, G., and Nesti, E. (2004b). A methodology for the derivation and verification of use cases for product lines. In *SPLC'04*, pages 255–265.

[Faulk, 2001] Faulk, S. R. (2001). Product-line requirements specification (PRS): an approach and case study. In *RE'01*, pages 48–55.

[Forbus and Kleer, 1993] Forbus, K. D. and Kleer, J. D. (1993). *Building Problem Solvers*. MIT Press.

[Gallina and Guelfi, 2007] Gallina, B. and Guelfi, N. (2007). A template for requirement elicitation of dependable product lines. In *REFSQ'07*, pages 63–77.

[Geppert et al., 2014] Geppert, B., Li, J., and Weiss, D. M. (2014). *Towards Generating Acceptance Tests for Product Lines*. Springer.

[Giese and Wagner, 2009] Giese, H. and Wagner, R. (2009). From model transformation to incremental bidirectional model synchronization. *Software and System Modeling*, 8:21–43.

[Goknil et al., 2013] Goknil, A., Kurtev, I., and Millo, J.-V. (2013). A metamodeling approach for reasoning on multiple requirements models. In *EDOC'13*, pages 159–166.

[Goknil et al., 2008a] Goknil, A., Kurtev, I., and van den Berg, K. (2008a). Change impact analysis based on formalization of trace relations for requirements. In *ECMDA-TW'08*, pages 59–75.

[Goknil et al., 2008b] Goknil, A., Kurtev, I., and van den Berg, K. (2008b). A metamodeling approach for reasoning about requirements. In *ECMDA-FA'08*, pages 310–325.

[Goknil et al., 2014] Goknil, A., Kurtev, I., van den Berg, K., and Spijkerman, W. (2014). Change impact analysis for requirements: A metamodeling approach. *Information and Software Technology*, 56(8):950 – 972.

[Goknil et al., 2011] Goknil, A., Kurtev, I., van den Berg, K., and Veldhuis, J.-W. (2011). Semantics of trace relations in requirements models for consistency checking and inferencing. *Software and Systems Modeling*, 10:31–54.

[Gomaa, 2000] Gomaa, H. (2000). Object oriented analysis and modeling families of systems with uml. In *ICSR-6*, pages 89–99.

[Gonzales-Sanchez et al., 2011] Gonzales-Sanchez, A., Piel, E., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2011). Prioritizing tests for software fault diagnosis. *Software Practice and Experience*, 41(19):1105–1129.

[Griss et al., 1998] Griss, M. L., Favaro, J., and d'Alessandro, M. (1998). Integrating feature modeling with the rseb. In *ICSR'98*, pages 76–85.

[Gurp et al., 2001] Gurp, J. V., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *WICSA'01*, pages 45–54.

[H. Cunningham et al, 2018] H. Cunningham et al (2018). Developing language processing components with gate version 8 (a user guide), `http://gate.ac.uk/sale/tao/tao.pdf`.

[Hajri et al., 2015] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2015). Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach. In *MODELS'15*, pages 338–347.

[Hajri et al., 2016] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2016). PUMConf: A tool to configure product specific use case and domain models in a product line. In *FSE'16*, pages 1008–1012.

[Hajri et al., 2017] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2017). Incremental reconfiguration of product specific use case models for evolving configuration decisions. In *REFSQ'17*, pages 1–19.

[Hajri et al., 2018a] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2018a). Change impact analysis for evolving configuration decisions in product line use case models. *Journal of Systems and Software*, 139:211–237.

[Hajri et al., 2018b] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2018b). Configuring use case models in product families. *Software and System Modeling*, 17(3):939–971.

[Halmans and Pohl, 2003] Halmans, G. and Pohl, K. (2003). Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2:15–36.

[Harel, 1987] Harel, D. (1987). Statecharts - a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

[Harrold et al., 2001] Harrold, M. J., Jones, J. A., Li, T., and Liang, D. (2001). Regression test selection for java software. In *OOPSLA'01*, pages 312–326.

[Hearnden et al., 2006] Hearnden, D., Lawley, M., and Raymond, K. (2006). Incremental model transformation for the evolution of model-driven systems. In *MODELS'06*, pages 321–335.

[Heider et al., 2012a] Heider, W., Rabiser, R., and Grünbacher, P. (2012a). Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. *International Journal on Software Tools for Technology Transfer*, 5:613–630.

[Heider et al., 2012b] Heider, W., Rabiser, R., Lettner, D., and Grünbacher, P. (2012b). Using regression testing to analyze the impact of changes to variability models on products. In *SPLC'12*, pages 196–205.

[Hein et al., 2000] Hein, A., Schlick, M., and Vinga-Martins, R. (2000). Applying feature models in industrial settings. In *SPLC'00*, pages 47–70.

[Hemmati et al., 2010] Hemmati, H., Briand, L., Arcuri, A., and Ali, S. (2010). An enhanced test case selection approach for model-based testing: An industrial case study. In *FSE'10*, pages 267–276.

[Hemmati et al., 2017] Hemmati, H., Fang, Z., Mantyla, M. V., and Adams, B. (2017). Prioritizing manual test cases in rapid release environments. *Software Testing, Verification and Reliability*, 27(6).

[Henard et al., 2014] Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., and Traon, Y. L. (2014). Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670.

[ISO, 2018] ISO (2018). ISO-26262: Road vehicles – functional safety.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. (1992). *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley.

[Jahann and Egyed, 2004] Jahann, S. and Egyed, A. (2004). Instant and incremental transformation of models. In *ASE'04*, pages 362–365.

[Jirapanthong and Zisman, 2009] Jirapanthong, W. and Zisman, A. (2009). Xtraque: traceability for product line systems. *SoSyM*, 8(1):117–144.

[Johansen et al., 2011] Johansen, M. F., Haugen, Ø., and Fleurey, F. (2011). A survey of empirics of strategies for software product line testing. In *ICSTW'11*, pages 266–269.

[John and Muthig, 2004] John, I. and Muthig, D. (2004). Product line modeling with generic use cases. In *EMPRESS'04*.

[Jr. et al., 2013] Jr., D. W. H., Lemeshow, S., and Sturdivant, R. X. (2013). *Applied Logistic Regression*. Wiley.

[Jurafsky and Martin, 2000] Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing*. Prentice Hall.

[Kahsai et al., 2008] Kahsai, T., Roggenbach, M., and Schlingloff, B.-H. (2008). Specification-based testing for software product lines. In *SEFM'08*, pages 149–159.

[Kamsties et al., 2004] Kamsties, E., Pohl, K., Reis, S., and Reuys, A. (2004). Testing variabilities in use case models. In *PFE 2003*, pages 6–18.

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon Software Engineering Institute.

[Kang et al., 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168.

[Kang and Lee, 2013] Kang, K. C. and Lee, H. (2013). Variability modeling. In *Systems and Software Variability Management, Concepts, Tools and Experiences*, pages 25–42. Springer.

[Karatas et al., 2010] Karatas, A. S., Oguztuzun, H., and Dogru, A. (2010). Mapping extended feature models to constraint logic programming over finite domains. In *SPLC'10*, pages 286–299.

[Kavakli, 2002] Kavakli, E. (2002). Goal-oriented requirements engineering: A unifying framework. *Requirements Engineering*, 6:237–251.

[Khatibsyarbini et al., 2018] Khatibsyarbini, M., Isa, M. A., Jawawi, D. N., and Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93.

[Knapp et al., 2014] Knapp, A., Roggenbach, M., and Schlingloff, B.-H. (2014). On the use of test cases in model-based software product line development. In *SPLC'14*, pages 247–251.

[Korel et al., 2008] Korel, B., Koutsogiannakis, G., and Tahat, L. H. (2008). Application of system models in regression test suite prioritization. In *ICSM'08*, pages 247–256.

[Korel et al., 2005] Korel, B., Tahat, L. H., and Harman, M. (2005). Test prioritization using system models. In *ICSM'05*, pages 559–568.

[Krishnamoorthi and Mary, 2009] Krishnamoorthi, R. and Mary, S. S. A. (2009). Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology*, 51:799–808.

[Kulak and Guiney, 2003] Kulak, D. and Guiney, E. (2003). *Use Cases: Requirements in Context*. Addison-Wesley.

[Kundu et al., 2009] Kundu, D., Sarma, M., Sarma, D., and Mall, R. (2009). System testing for object-oriented systems with test case prioritization. *Software Testing, Verification and Reliability*, 19(4):297–333.

[Kung et al., 1995] Kung, D. C., Gao, J., and Hsia, P. (1995). Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65.

[Kurtev et al., 2007] Kurtev, I., Dee, M., Göknil, A., and van den Berg, K. (2007). Traceability-based change management in operational mappings. In *ECMDA-TW'07*, pages 57–67.

[Lachmann et al., 2017] Lachmann, R., Beddig, S., Lity, S., Schulze, S., and Schaefer, I. (2017). Risk-based integration testing of software product lines. In *VaMoS'17*, pages 52–59.

[Lachmann et al., 2016a] Lachmann, R., Lity, S., Al-Hajjaji, M., Furchtegott, F., and Schaefer, I. (2016a). Fine-grained test case prioritization for integration testing of delta-oriented software product lines. In *FOSD'16*.

[Lachmann et al., 2015] Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., and Schaefer, I. (2015). Delta-oriented test case prioritization for integration testing of software product lines. In *SPLC'15*, pages 81–90.

[Lachmann et al., 2016b] Lachmann, R., Nieke, M., Seidl, C., Schaefer, I., and Schulze, S. (2016b). System-level test case prioritization using machine learning. In *ICMLA'16*, pages 361–368.

[Larman, 2002] Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall.

[Lauenroth and Pohl, 2007] Lauenroth, K. and Pohl, K. (2007). Towards automated consistency checks of product line requirements specifications. In *ASE'07*, pages 373–376.

[Lauenroth and Pohl, 2008] Lauenroth, K. and Pohl, K. (2008). Dynamic consistency checking of domain requirements in product line engineering. In *RE'08*, pages 193–202.

[Lauesen, 2002] Lauesen, S. (2002). *Requirements - Styles and Techniques*. Addison-Wesley.

[Lee et al., 2012] Lee, J., Kang, S., and Lee, D. (2012). A survey on software product line testing. In *SPLC'12*, pages 31–40.

[Li et al., 2007] Li, Z., Harman, M., and Hierons, R. M. (2007). Search algorithms for regression test prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237.

[Lity et al., 2017] Lity, S., Al-Hajjaji, M., Thum, T., and Schaefer, I. (2017). Optimizing product orders using graph algorithms for improving incremental product-line analysis. In *VaMoS'17*, pages 60–67.

[Lity et al., 2012] Lity, S., Lochau, M., Schaefer, I., and Goltz, U. (2012). Delta-oriented model-based spl regression testing. In *PLEASE'12*, pages 53–56.

[Lity et al., 2016] Lity, S., Morbach, T., Thum, T., and Schaefer, I. (2016). Applying incremental model slicing to product-line regression testing. In *ICSR'16*, pages 3–19.

[Lochau et al., 2014] Lochau, M., Lity, S., Lachmann, R., Schaefer, I., and Goltz, U. (2014). Delta-oriented model-based integration testing of large-scale systems. *Journal of Systems and Software*, 91:63–84.

[Manning et al., 2008] Manning, C. D., Raghavan, P., and Schutze, H. (2008). *Introduction to Information Introduction to Information Retrieval*. Cambridge University Press.

[Mannion, 2002] Mannion, M. (2002). Using first-order logic for product line model validation. In *SPLC'02*, pages 176–187.

[Mannion and Camara, 2003] Mannion, M. and Camara, J. (2003). Theorem proving for product line model verification. In *PFE'03*, pages 211–224.

[McGregor, 2001] McGregor, J. D. (2001). Testing a software product line. Technical report, Carnegie Mellon Software Engineering Institute.

[Mendonca et al., 2009a] Mendonca, M., Branco, M., and Cowan, D. (2009a). S.P.L.O.T. - software product lines online tools. In 761-762, editor, *OOPSLA'09*.

[Mendonca et al., 2009b] Mendonca, M., Wasowski, A., and Czarnecki, K. (2009b). SAT-based analysis of feature models is easy. In *SPLC'09*, pages 231–240.

[Mirarab et al., 2008] Mirarab, S., Ganjali, A., Tahvildari, L., Li, S., Liu, W., and Morrissey, M. (2008). A requirement-based software testing framework: An industrial practice. In *ICSM'08*, pages 452–455.

[Moon and Yeom, 2004] Moon, M. and Yeom, K. (2004). An approach to develop requirement as a core asset in product line. In *ICSR'04*, pages 23–34.

[Moon et al., 2005] Moon, M., Yeom, K., and Chae, H. S. (2005). An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *IEEE Transactions on Software Engineering*, 31(7):551–569.

[Muccini, 2007] Muccini, H. (2007). Using model differencing for architecture-level regression testing. In *SEAA'07*.

[Muccini et al., 2006] Muccini, H., Dias, M., and Richardson, D. J. (2006). Software architecture-based regression testing. *Journal of Systems and Software*, 79:1379–1396.

[Muccini and van der Hoek, 0382] Muccini, H. and van der Hoek, A. (2003§82). Towards testing product line architectures. *Electronic Notes in Theoretical Computer Science*, 6(99–109).

[Mussbacher et al., 2012] Mussbacher, G., Araújo, J., Moreira, A., and Amyot, D. (2012). AoURN-based modeling and analysis of software product lines. *Software Quality Journal*, 20:645–687.

[Myllärniemi et al., 2005] Myllärniemi, V., Asikainen, T., Männistö, T., and Soininen, T. (2005). Kumbang configurator - a configuration tool for software product families. In *IJCAI-05*, pages 51–57.

[Nardo et al., 2015] Nardo, D. D., Alshahwan, N., Briand, L., and Labiche, Y. (2015). Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396.

[Nebut et al., 2006] Nebut, C., Traon, Y. L., and Jezequel, J.-M. (2006). System testing of product families: from requirements to test cases. In *Software Product Lines*. Springer.

[Nejati et al., 2016] Nejati, S., Sabetzadeh, M., Arora, C., Briand, L. C., and Mandoux, F. (2016). Automated change impact analysis between SysML models of requirements and design. In *FSE'16*, pages 242–253.

[Nie et al., 2013] Nie, K., Yue, T., Ali, S., Zhang, L., and Fan, Z. (2013). Constraints: The core of supporting automated product configuration of cyber-physical systems. In *MODELS'13*, pages 370–387.

[Nielsen, 1994] Nielsen, J. (1994). Usability inspection methods. In *CHI '94*, pages 413–414.

[Nielsen and Molich, 1990] Nielsen, J. and Molich, R. (1990). Heuristic evaluation of user interfaces. In *CHI '90*, pages 249–256.

[Nöhrer et al., 2012] Nöhrer, A., Biere, A., and Egyed, A. (2012). Managing SAT inconsistencies with HUMUS. In *VaMoS'12*, pages 83–91.

[Nöhrer and Egyed, 2010] Nöhrer, A. and Egyed, A. (2010). Conflict resolution strategies during product configuration. In *VaMoS'10*, pages 107–114.

[Nöhrer and Egyed, 2013] Nöhrer, A. and Egyed, A. (2013). C2O configurator: a tool for guided decision-making. *Automated Software Engineering*, 20:265–296.

[Nuseibeh and Easterbrook, 2000] Nuseibeh, B. and Easterbrook, S. (2000). Requirements engineering: A roadmap. In *ICSE'00*, pages 35–46.

[Oppenheim, 2005] Oppenheim, A. N. (2005). *Questionnaire Design, Interviewing and Attitude Measurement*. Continuum.

[Oster et al., 2011] Oster, S., Wübbeke, A., Engels, G., and Schürr, A. (2011). A survey of model-based software product lines testing. *Model-Based Testing for Embedded Systems*, pages 338–381.

[Parejo et al., 2016] Parejo, J. A., Sánchez, A. B., Segura, S., Ruiz-Cortés, A., Lopez-Herrejon, R. E., and Egyed, A. (2016). Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software*, 122:287–310.

[Paskevicius et al., 2012] Paskevicius, P., Damasevicius, R., and Štuikys, V. (2012). Change impact analysis of feature models. In *ICIST'12*, pages 108–122.

[Passos et al., 2013] Passos, L., Czarnecki, K., Apel, S., Wasowski, A., Kästner, C., and Guo, J. (2013). Feature-oriented software evolution. In *VaMoS'13*.

[Pohl, 2010] Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition.

[Pohl et al., 2005] Pohl, K., Bockle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.

[Pohl and Haumer, 1997] Pohl, K. and Haumer, P. (1997). Modelling contextual information about scenarios. In *REFSQ'97*, pages 187–204.

[Pohl and Metzger, 2006] Pohl, K. and Metzger, A. (2006). Software product line testing. *Communications of the ACM*, 49(12):78–81.

[Pradhan et al., 2018] Pradhan, D., Wang, S., Ali, S., Yue, T., and Liaaen, M. (2018). REMAP: Using rule mining and multi-objective search for dynamic test case prioritization. In *ICST'18*, pages 46–57.

[Qu et al., 2011] Qu, X., Acharya, M., and Robinson, B. (2011). Impact analysis of configuration changes for test case selection. In *ISSRE'11*, pages 140–149.

[Quinton et al., 2015] Quinton, C., Rabiser, R., Vierhauser, M., Grunbacher, P., and Baresi, L. (2015). Evolution in dynamic software product lines: Challenges and perspectives. In *SPLC'15*, pages 126–130.

[Rabiser et al., 2010] Rabiser, R., Grünbacher, P., and Dhungana, D. (2010). Requirements for product derivation support: Results from a systematic literature review. *Information and Software Technology*, 52:324–346.

[Ramesh and Jarke, 2001] Ramesh, B. and Jarke, M. (2001). Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93.

[Reuys et al., 2005] Reuys, A., Kamsties, E., Pohl, K., and Reis, S. (2005). Model-based system testing of software product families. In *CAiSE'05*, pages 519–534.

[Reuys et al., 2006] Reuys, A., Reis, S., Kamsties, E., and Pohl, K. (2006). The ScenTED method for testing software product lines. In *Software Product Lines*, pages 479–520.

[Rice, 2007] Rice, J. A. (2007). *Mathematical Statistics and Data Analysis*. Thomson Higher Education.

[Riebisch et al., 2002] Riebisch, M., Bollert, K., Streitferdt, D., and Philippow, I. (2002). Extending feature diagrams with uml multiplicities. In *IDPT'02*.

[Robertson and Robertson, 2006] Robertson, S. and Robertson, J. (2006). *Mastering the Requirements Process*. Addison-Wesley.

[Rosa et al., 2009] Rosa, M. L., van der Aalst, W. M. P., Dumas, M., and ter Hofstede, A. H. M. (2009). Questionnaire-based variability modeling for system configuration. *Software and Systems Modeling*, 8:251–274.

[Rothermel and Harrold, 1996] Rothermel, G. and Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551.

[Rothermel and Harrold, 1997] Rothermel, G. and Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210.

[Rothermel et al., 2000] Rothermel, G., Harrold, M. J., and Dedhia, J. (2000). Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109.

[Rothermel et al., 2001] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948.

[RTCA and EUROCAE, 2018] RTCA and EUROCAE (2018). DO-178C: Software considerations in airborne systems and equipment certification.

[Runeson and Engström, 2012] Runeson, P. and Engström, E. (2012). Regression testing in software product line engineering. In *Advances in Computers*, volume 86, pages 223–263.

[Rus et al., 2013] Rus, V., Lintean, M., Banjade, R., Niraula, N., and Stefanescu, D. (2013). SEMILAR: The semantic similarity toolkit. In *ACL'13*, pages 163–168.

[Seidl et al., 2012] Seidl, C., Heidenreich, F., and Aßmann, U. (2012). Co-evolution of models and feature mapping in software product lines. In *SPLC'12*, pages 76–85.

[seok Choi et al., 2008] seok Choi, W., Kang, S., Choi, H., and Baik, J. (2008). Automated generation of product use case scenarios in product line development. In *CIT'08*, pages 760–765.

[Shurr et al., 2010] Shurr, A., Oster, S., and Markert, F. (2010). Model-driven software product lines testing: An integrated approach. In *SOFSEM'10*, pages 112–131.

[Sinnema and Deelstra, 2008] Sinnema, M. and Deelstra, S. (2008). Industrial validation of COVAMOF. *Journal of Systems and Software*, 81:584–600.

[Sinnema et al., 2004] Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J. (2004). COVAMOF: A framework for modeling variability in software product families. In *SPLC'04*, pages 197–213.

[Smith and N.Mosier, 1986] Smith, S. L. and N.Mosier, J. (1986). *Guidelines for designing user interface software*.

[Sokolova and Lapalme, 2009] Sokolova, M. and Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437.

[Sommerville, 2009] Sommerville, I. (2009). *Software Engineering (9th ed.)*. Addison-Wesley.

[Srikanth and Banerjee, 2012] Srikanth, H. and Banerjee, S. (2012). Improving test efficiency through system test prioritization. *Journal of Systems and Software*, 85:1176–1187.

[Srikanth et al., 2016] Srikanth, H., Hettiarachchi, C., and Do, H. (2016). Requirements based test prioritization using risk factors: An industrial study. *Information and Software Technology*, 69:71–83.

[Srikanth and Williams, 2005] Srikanth, H. and Williams, L. (2005). On the economics of requirements-based test case prioritization. In *EDSER'05*, pages 1–3.

[Srikanth et al., 2005] Srikanth, H., Williams, L., and Osborne, J. (2005). System test case prioritization of new and regression test cases. In *ESEM'05*, pages 64–73.

[Srikanth et al., 2014] Srikanth, H., Williams, L., and Osborne, J. (2014). Towards the prioritization of system test cases. *Software Testing, Verification and Reliability*, pages 320–337.

[Stoiber, 2012] Stoiber, R. (2012). *A New Approach to Product Line Engineering in Model-Based Requirements Engineering*. PhD thesis, University of Zurich.

[Stoiber et al., 2010] Stoiber, R., Fricker, S., Jehle, M., and Glinz, M. (2010). Feature unweaving: Refactoring software requirements specifications into software product lines. In *RE'10*, pages 403–404.

[Stoiber and Glinz, 2010] Stoiber, R. and Glinz, M. (2010). Supporting stepwise, incremental product derivation in product line requirements engineering. In *VaMoS'10*, pages 77–84.

[Stricker et al., 2010] Stricker, V., Metzger, A., and Pohl, K. (2010). Avoiding redundant testing in application engineering. In *SPLC'10*, pages 226–240.

[Sun et al., 2005] Sun, J., Zhang, H., Li, Y. F., and Wang, H. (2005). Formal semantics and verification for feature modeling. In *ICECCS'05*.

[Tahat et al., 2012] Tahat, L., Korel, B., Harman, M., and Ural, H. (2012). Regression test suite prioritization using system models. *Software Testing, Verification and Reliability*, 22(7):481–506.

[ten Hove et al., 2009] ten Hove, D., Goknil, A., Kurtev, I., van den Berg, K., and de Goede, K. (2009). Change impact analysis for SysML requirements models based on semantics of trace relations. In *ECMDA-TW'09*, pages 17–28.

[Tevanlinna et al., 2004] Tevanlinna, A., Taina, J., and Kauppinen, R. (2004). Product family testing: a survey. *ACM SIGSOFT Software Engineering Notes*, 29(2):12–22.

[Thüm et al., 2009] Thüm, T., Batory, D. S., and Kästner, C. (2009). Reasoning about edits to feature models. In *ICSE'09*, pages 254–264.

[Tonella et al., 2006] Tonella, P., Avesani, P., and Susi, A. (2006). Using the case-based ranking methodology for test case prioritization. In *ICSM'06*, pages 123–133.

[Trinidad et al., 2008] Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., and Toro, M. (2008). Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81:883–896.

[Trinidad and Ruiz-Cortés, 2009] Trinidad, P. and Ruiz-Cortés, A. (2009). Abductive reasoning and automated analysis of feature models: How are they connected? In *VaMoS'09*, pages 145–153.

[Uzuncaova et al., 2008] Uzuncaova, E., Garcia, D., Khurshid, S., and Batory, D. S. (2008). Testing software product lines using incremental test generation. In *ISSRE'08*, pages 249–258.

[Uzuncaova et al., 2010] Uzuncaova, E., Khurshid, S., and Batory, D. S. (2010). Incremental test generation for software product lines. *IEEE Transactions on Software Engineering*, 36(3):309–322.

[van der Linden, 2002] van der Linden, F. (2002). Software product families in europe: The ESAPS and CAFE projects. *IEEE Software*, 19(4):41–49.

[van Lamsweerde, 2001] van Lamsweerde, A. (2001). Goal-oriented requirements engineering - a guided tour. In *RE'01*, pages 249–263.

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley & Sons.

[Varela et al., 2011] Varela, P., Araújo, J., Brito, I., and Moreira, A. (2011). Aspect-oriented analysis for software product lines requirements engineering. In *SAC'11*, pages 667–674.

[Vaysburg et al., 2002] Vaysburg, B., Tahat, L. H., and Korel, B. (2002). Dependence analysis in reduction of requirement based test suites. In *ISSTA'02*, pages 107–111.

[Vogel et al., 2009] Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., and Becker, B. (2009). Incremental model synchronization for efficient run-time monitoring. In *MODELS Workshops 2009*, pages 124–139.

[von der Maßen and Lichter, 2002] von der Maßen, T. and Lichter, H. (2002). Modeling variability by uml use case diagrams. In *REPL'02*, pages 19–25.

[von Mayrhauser and Zhang, 1999] von Mayrhauser, A. and Zhang, N. (1999). Automated regression testing using dbt and sleuth. *Journal of Software Maintenance*, 11(2):93–116.

[Wang et al., 2009] Wang, B., Zhang, W., Zhao, H., Jin, Z., and Mei, H. (2009). A use case based approach to feature models' construction. In *RE'09*, pages 121–130.

[Wang et al., 2015a] Wang, C., Pastore, F., Goknil, A., Briand, L. C., and Iqbal, M. Z. Z. (2015a). Automatic generation of system test cases from use case specifications. In *ISSTA'15*, pages 385–396.

[Wang et al., 2015b] Wang, C., Pastore, F., Goknil, A., Briand, L. C., and Iqbal, M. Z. Z. (2015b). UMTG: a toolset to automatically generate system test cases from use case specifications. In *ESEC/SIGSOFT FSE'15*, pages 942–945.

[Wang et al., 2005] Wang, H., Li, Y. F., Sun, J., Zhang, H., and Pan, J. (2005). A semantic web approach to feature modeling and verification. In *SWESE'05*.

[Wang et al., 2007] Wang, H., Li, Y. F., Zhang, H., and Pan, J. (2007). Verifying feature models using owl. *Journal of Web Semantics*, 5:117–129.

[Wang et al., 2016] Wang, S., Ali, S., Gotlieb, A., and Liaaen, M. (2016). A systematic test case selection methodology for product lines: Results and insights from an industrial case study. *Empirical Software Engineering*, 21(4):1586–1622.

[Wang et al., 2017] Wang, S., Ali, S., Gotlieb, A., and Liaaen, M. (2017). Automated product line test case selection: Industrial case study and controlled experiment. *Software and Systems Modeling*, 16(2):417–441.

[Wang et al., 2014] Wang, S., Buchmann, D., Ali, S., Gotlieb, A., Pradhan, D., and Liaaen, M. (2014). Multi-objective test prioritization in software product line testing: An industrial case study. In *SPLC'14*, pages 32–41.

[Wang et al., 2013] Wang, S., Gotlieb, A., Ali, S., and Liaaen, M. (2013). Automated test case selection using feature model: An industrial case study. In *MODELS'13*, pages 237–253.

[Weiss and Lai, 1999] Weiss, D. M. and Lai, C. T. R. (1999). *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley.

[Weston et al., 2009] Weston, N., Chitchyan, R., and Rashid, A. (2009). A framework for constructing semantically composable feature models from natural language requirements. In *SPLC'09*, pages 211–220.

[White et al., 2010] White, J., Benavides, D., Schmidt, D. C., Trinidad, P., Dougherty, B., and Ruiz-Cortés, A. (2010). Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83:1094–1107.

[White et al., 2014] White, J., Galindo, J. A., Saxena, T., Dougherty, B., Benavides, D., and Schmidt, D. C. (2014). Evolving feature model configurations in software product lines. *Journal of Systems and Software*, pages 119–136.

[White et al., 2008] White, J., Schmidt, D. C., Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2008). Automated diagnosis of product-line configuration errors in feature models. In *SPLC'08*, pages 225–234.

[Wiegers and Beatty, 2013] Wiegers, K. and Beatty, J. (2013). *Software Requirements*. Microsoft Press.

[Wieringa, 2009] Wieringa, R. (2009). Design science as a nested problem solving. In *DESRIST'09*.

[Wieringa, 2010] Wieringa, R. (2010). Relevance and problem choice in design science. In *DESRIST'10*, pages 61–76.

[Wieringa et al., 2006] Wieringa, R., Maiden, N. A. M., Mead, N. R., and Rolland, C. (2006). Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering Journal*, 11(1):102–107.

[Wohlin et al., 2012] Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., and Wesslen, A. (2012). *Experimentation in Software Engineering*. Springer.

[Wong et al., 1997] Wong, W. E., Horgan, J. R., London, S., and Bellcore, H. A. (1997). A study of effective regression testing in practice. In *ISSRE'97*, pages 230–238.

[Xiong et al., 2007] Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., and Mei, H. (2007). Towards automatic model synchronization from model transformations. In *ASE'07*, pages 164–173.

[Yoo and Harman, 2012] Yoo, S. and Harman, M. (2012). Regression testing minimisation, selection and prioritisation : A survey. *Software Testing, Verification and Reliability*, 22(2):67–120.

[Yu, 1997] Yu, E. S. K. (1997). Towards modeling and reasoning support for early-phase requirements engineering. In *RE'97*, pages 226–235.

[Yue et al., 2011] Yue, T., Ali, S., and Briand, L. C. (2011). Automated transition from use cases to uml state machines to support state-based testing. In *ECMFA'11*, pages 115–131.

[Yue et al., 2015a] Yue, T., Ali, S., and Zhang, M. (2015a). RTCM: a natural language based, automated, and practical test case generation framework. In *ISSTA'15*, pages 397–408.

[Yue et al., 2013] Yue, T., Briand, L. C., and Labiche, Y. (2013). Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology*, 22(1).

[Yue et al., 2015b] Yue, T., Briand, L. C., and Labiche, Y. (2015b). aToucan: An automated framework to derive uml analysis models from use case models. *ACM Transactions on Software Engineering and Methodology*, 24(3).

[Zech et al., 2017] Zech, P., Kalb, P., Felderer, M., Atkinson, C., and Breu, R. (2017). Model-based regression testing by ocl. *Software Tools for Technology Transfer*, 19(1):115–131.

[Zhang et al., 2013] Zhang, G., Yue, T., Wu, J., and Ali, S. (2013). Zen-RUCM: A tool for supporting a comprehensive and extensible use case modeling framework. In *Demos@MoDELS 2013*, pages 41–45.

[Ziadi and Jezequel, 2006] Ziadi, T. and Jezequel, J.-M. (2006). Product line engineering with the uml: Deriving products. In *Software Product Lines*. Springer.

[Zschaler et al., 2009] Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., and Kulesza, U. (2009). Vml* – a family of languages for variability management in software product lines. In *SLE'09*, pages 82–102.

Appendix A

Configuration Algorithm

In this appendix, we provide the algorithm for the configuration of Product Specific (PS) use case and domain models. The algorithm is presented in seven functions. The main function, config, is given in Algs. 11 and 12. It collects the configuration decisions from the user and checks if they are consistent. Decisions made by the user are said to be consistent if they do not violate any dependency constraint (i.e., constraints imposed by the *require* and *conflict* relations specified in the Product Line use case diagram). After all decisions are collected and confirmed to be consistent, the algorithm generates the PS use case and domain models (the function calls for *generatePSDiagram*, *generatePSSpecifications* and *generatePSDomainModel* in Algs. 11 and 12). Six functions (Algs. 25-22) are called in config to check decision consistency:

- checkConflictingVP (Alg. 25) and checkConflictingUC (Alg. 23) identify the contradicting decisions with regards to the *conflict* relation for selected variation points and variant use cases;
- checkRequiringVP (Alg. 15) and checkRequiringUC (Alg. 27) identify the contradicting decisions with regards to the *require* relation for selected variation points and variant use cases, respectively;
- checkRequiredVP (Alg. 16) and checkRequiredUC (Alg. 22) determine the contradicting decisions with regards to the *require* relation for unselected variation points and variant use cases, respectively;

The check functions calls some other functions (Alg. 19 and Alg. 20) in order to infer use cases which are automatically selected or unselected.

- inferUnselectedElements (Alg. 19) infers unselected elements for the use case unselected in the given decision. Inferred, unselected use cases are in the variation points included by the unselected use cases. All the variant use cases of the inferred, unselected variation points are unselected and have no more further decisions to be made.
- inferSelectedElements (Alg. 20) infers selected elements for the use case selected in the given decision. Inferred use cases are with a mandatory variability relation in the variation points

included by the selected use cases. Inferred, selected variation points are either variation points which have inferred, selected use cases, or mandatory variation points included by inferred, selected use cases.

Alg. 11: config

```
Inputs : Product Line use case diagram, $PLD$,
         Set of Product Line use case specifications, $PLS$,
         Product Line domain model, $PLDM$
Output: Set of Product Specific use case and domain models
Let $PSD$ be the product specific use case diagram
Let $PSS$ be the product specific use case specifications
Let $PSDM$ be the product specific domain model
Let $PSM$ be the set of product specific use case and domain models
Let $DC$ be the empty set for completed decisions for $PLD$
Let $L$ be the set of pairs of variation points $vp$ and use cases $uc$ such that use cases are essential and they include the variation points, or the variation points are not included by any use case
$T \leftarrow L$
while $L \neq \emptyset$ do
    $dp \in L$
    Let $SUC$ be the set of variant use cases selected in $dp.vp$
    Let $NSUC$ be the set of variant use cases unselected in $dp.vp$
    Let $d$ be the quadruple $(dp.vp, dp.uc, SUC, NSUC)$
    if (d satisfies the cardinality constraints in dp.vp) then
        Let $C$ be the empty set for contradicting decisions
        $C \leftarrow$ checkConflictingVP($dp.vp$, $DC$, $d$, $PLD$)
        $C \leftarrow C \cup$ checkRequiringVP($dp.vp$, $DC$, $d$, $PLD$)
        $C \leftarrow C \cup$ checkRequiredVP($dp.vp$, $DC$, $d$, $PLD$)
        foreach ($u \in SUC$) do
            $C \leftarrow C \cup$ checkConflictingUC($u$, $DC$, $d$, $PLD$)
            $C \leftarrow C \cup$ checkRequiringUC($u$, $DC$, $d$, $PLD$)
        end foreach
        foreach ($u \in NSUC$) do
            $C \leftarrow C \cup$ checkRequiredUC($u$, $DC$, $d$, $PLD$)
        end foreach
        if ($C = \emptyset$) then
            $DC \leftarrow DC \cup \{d\}$;
            $L \leftarrow L \backslash \{dp\}$
            Let $new_p = \{(vp, uc) \mid uc\,\textbf{includes}\,vp \wedge uc \in SUC \wedge (vp, uc) \notin T\}$
            $L \leftarrow L \cup new_p$
            $T \leftarrow T \cup new_p$
        else
            updateDecisions($C \cup \{d\}$)
        end if
    else
        updateDecisions($\{d\}$)
    end if
end while
$C \leftarrow$ overallConsistencyCheck($DC, PLD$)
if ($C = \emptyset$) then
    $PSD \leftarrow$ generatePSDiagram($DC$, $PLD$)
else
    do
        updateDecisions($C$)
        $C \leftarrow$ overallConsistencyCheck($DC, PLD$)
    while ($C \neq \emptyset$)
    $PSD \leftarrow$ generatePSDiagram($DC$, $PLD$)
end if
```

Alg. 12: config(continued)

```
/* Configure PS Use Case Specifications*/
Let $U$ be the set of use cases selected in $DC$ containing variability in their specifications
Let $Y$ be the set of essential use cases containing variability in their specifications
$F \leftarrow U \cup Y$
Let $S$ be the empty set for optional steps and alternative flows selected by the user
Let $O$ be the empty set for decided orders
foreach ($fc \in F$) do
    Let $OF$ be the set of selected, optional alternative flows, of $fc$, containing variability
    Let $b$ be the basic flow, of $fc$, containing variability
    Let $AF$ be the set of mandatory alternative flows, of $fc$, containing variability
    $AC \leftarrow OF \cup \{b\} \cup \{AF\}$
    foreach ($a \in AC$) do
        Let $MS$ be the set of mandatory steps in $a$
        Let $SS$ be the set of optional steps selected by the user from $a$
        Let $ORD$ be the set of steps that need to be ordered in $SS \cup MS$
        if ($ORD \neq \emptyset$) then
            $O \leftarrow O \cup$ updateStepOrder($ORD$)
        end if
        $S \leftarrow S \cup SS \cup OF \cup O$
    end foreach
end foreach
$PSS \leftarrow$ generatePSSpecifications($S$, $PLD$, $PLS$)
/* Configure PS Domain Model*/
Let $DE$ be the empty set for optional and variant entities selected by the user
Let $OE$ be the set of selected, optional domain entities in $PLDM$
$DE \leftarrow DE \cup OE$
Let $V$ be the set of variation domain entities in $PLDM$
foreach ($vc \in V$) do
    Let $VE$ be the set of selected, variant domain entities in $vc$
    $DE \leftarrow DE \cup VE$
end foreach
$PSDM \leftarrow$ generatePSDomainModel($DE$, $PLDM$)
/* Save all the decisions in the decision model and return the generated models*/
Let $DM$ be the decision model for the configuration
$DM \leftarrow$ saveDecisions($DC$, $S$, $O$, $DE$)
$PSM \leftarrow \{PSD\} \cup \{PSS\} \cup \{PSDM\} \cup \{DM\}$
return $PSM$
```

Alg. 13: checkConflictingVP

```
Input  : Variation point, $vp$, Set of decisions, $DC$,
         Decision, $d$, Product Line use case diagram, $PLD$
Output: Set of contradicting decisions
Let $RES$ be the empty set for contradicting decisions
Let $SUC$ be the set of selected variant use cases of variation point $vp$ in decision $d$/* d.SUC in the quadruple*/
if ($SUC \neq \emptyset$) then
    Let $C$ be the empty set for conflicting elements
    $C \leftarrow$ getConflictingElements($vp$, $PLD$) /* get set of elements that conflict with vp*/
    if ($C \neq \emptyset$) then
        foreach ($dm \in DC$) do
            Let $p$ be the variation point in decision $dm$ /* dm.vp in the quadruple*/
            Let $S$ be the set of selected use cases of variation point $p$ in decision $dm$ /* dm.SUC in the quadruple*/
            if ($S \neq \emptyset$) and ($p \in C$)) then
                $RES \leftarrow RES \cup \{dm\}$
            end if
            foreach ($u \in S$) do
                Let $I$ be the empty set for inferred, selected elements
                $I \leftarrow \{u\} \cup$ inferSelectedElements($u$, $dm$, $DC$, $\emptyset$, $PLD$)
                foreach ($e \in I$) do
                    if ($e \in C$) then
                        $RES \leftarrow RES \cup$ getInvolvedDecisions($e$, $dm$, $d$, $DC$)
                    end if
                end foreach
            end foreach
        end foreach
    end if
end if
return $RES$
```

Alg. 14: checkConflictingUC

```
Input  : Use Case, $uc$, Set of decisions, $DC$,
         Decision, $d$, Product Line use case diagram, $PLD$
Output: Set of contradicting decisions

Let $RES$ be the empty set for contradicting decisions
Let $I$ be the empty set for inferred, selected elements
$I \leftarrow \{uc\} \cup$ inferSelectedElements($uc$, $d$, $DC$, $\emptyset$, $PLD$)
Let $C$ be the empty set for conflicting elements
foreach ($e \in I$) do
    $C \leftarrow C \cup$getConflictingElements($e$, $PLD$) /* get the set of elements conflicting with e*/
end foreach
if ($C \neq \emptyset$) then
    foreach ($dm \in DC$) do
        Let $vp$ be the variation point in decision $dm$ /* dm.vp in the quadruple*/
        Let $SUC$ be the set of selected variant use cases in $dm$ /* dm.SUC in the quadruple*/
        if $((SUC \notin \emptyset)$ and $(vp \in C))$ then
            $RES \leftarrow RES \cup \{dm\}$
        end if
        foreach ($u \in SUC$) do
            Let $B$ be the empty set for inferred, selected elements
            $B \leftarrow \{u\} \cup$ inferSelectedElements($u$, $dm$, $DC$, $\emptyset$, $PLD$)
            foreach ($e \in B$) do
                if ($e \in C$) then
                    $RES \leftarrow RES \cup$ getInvolvedDecisions($e$, $dm$, $d$, $DC$)
                end if
            end foreach
        end foreach
    end foreach
end if
return $RES$
```

Alg. 15: checkRequiringVP

```
Input  : Variation Point, $vp$, Set of decisions, $DC$,
         Decision, $d$, Product Line use case diagram, $PLD$
Output: Set of contradicting decisions

Let $RES$ be the empty set for contradicting decisions
Let $R$ be the empty set for required elements
Let $SUC$ be the set of selected variant use cases in $d$  /* d.SUC in the quadruple*/
if ($SUC \neq \emptyset$) then
    $R \leftarrow$ getRequiredElements($vp$, $PLD$)
    if ($R \neq \emptyset$) then
        foreach ($dm \in DC$) do
            Let $p$ be the variation point in the decision $dm$  /* dm.vp in the quadruple*/
            if ((p is unselected in $DC \cup \{d\}$) and (there is no more decision to be made for p)
            and ($p \in R$)) then
                $RES \leftarrow RES \cup$ getInvolvedDecisions($p$, $dm$, $d$, $DC$)
            end if
            Let $NSUC$ be the set of unselected variant use cases in $dm$  /* dm.NSUC in the
            quadruple*/
            foreach ($v \in NSUC$) do
                Let $EX$ be the empty set for inferred, unselected elements
                $EX \leftarrow \{v\} \cup$ inferUnselectedElements($v$, $d$, $DC$, $\emptyset$, $PLD$)
                foreach ($e \in EX$) do
                    if ((e is unselected in $DC \cup \{d\}$) and (there is no more decision to be made for
                    e) and ($e \in R$)) then
                        $RES \leftarrow RES \cup$ getInvolvedDecisions($e$, $dm$, $d$, $DC$)
                    end if
                end foreach
            end foreach
        end foreach
    end if
end if
return $RES$
```

Alg. 16: checkRequiredVP

```
Input  : Variation Point, vp, Set of decisions, DC,
         Decision, d, Product Line use case diagram, PLD
Output: Set of contradicting decisions

Let RES be the empty set for contradicting decisions
Let R be the empty set for required elements
if ((vp is unselected in $DC \cup \{d\}$) and (there is no more decision to be made for vp)) then
    foreach ($dm \in DC$) do
        $R \leftarrow$ getRequiredElements(dm.vp, PLD)
        Let SUC be the set of selected elements in dm /* dm.SUC in the quadruple*/
        if (($SUC \neq \emptyset$) and ($vp \in R$)) then
            $RES \leftarrow RES \cup \{dm\}$
        end if
        foreach ($u \in SUC$) do
            Let I be the set of inferred, selected elements (variant use cases and variation points)
            $I \leftarrow \{u\} \cup$ inferSelectedElements(u, dm, $DC \cup \{d\}$, $\emptyset$, PLD)
            foreach ($e \in I$) do
                if ($vp \in$ getRequiredElements(e, PLD)) then
                    $RES \leftarrow RES \cup$ getInvolvedDecisions(e, dm, d, DC)
                end if
            end foreach
        end foreach
    end foreach
end if
return RES
```

Alg. 17: checkRequiringUC

```
Input  : Use Case, $uc$, Set of decisions, $DC$,
         Decision, $d$, Product Line use case diagram, $PLD$
Output: Set of contradicting decisions

Let $RES$ be the empty set for contradicting decisions
Let $I$ be the empty set for inferred, selected elements (variant use cases and variation points)
$I \leftarrow \{uc\} \cup$ inferSelectedElements($uc$, $d$, $DC$, $\emptyset$, $PLD$)
Let $R$ be the empty set for required elements
foreach ($e \in I$) do
    $R \leftarrow R \cup$ getRequiredElements($e$, $PLD$) /* get set of elements required by e*/
end foreach
if ($R \neq \emptyset$) then
    foreach ($dm \in DC$) do
        Let $vp$ be the the variation point in $dm$ /* dm.vp in the quadruple*/
        if (($vp$ is unselected in $DC \cup \{d\}$) and (there is no more decision to be made for $vp$) and
        ($vp \in R$)) then
            $RES \leftarrow RES \cup$ getInvolvedDecisions($vp$, $dm$, $d$, $DC$)
        end if
        Let $NSUC$ be the set of unselected elements in $dm$ /* dm.NSUC in the quadruple*/
        foreach ($n \in NSUC$) do
            if (($n$ is not selected in $DC \cup \{d\}$) and (there is no more decision to be made for $n$)
            and ($n \in R$)) then
                $RES \leftarrow RES \cup$ getInvolvedDecisions($n$, $dm$, $d$, $DC$)
            end if
        end foreach
    end foreach
end if
return $RES$
```

Alg. 18: checkRequiredUC

```
Input  : Use Case, uc, Set of decisions, DC,
         Decision, d, Product Line use case diagram, PLD
Output: Set of contradicting decisions
Let RES be the empty set for contradicting decisions
if ((uc is not selected in the completed decisions DC) and (there is no further decision to be made for uc)) then
    Let EX be the empty set for inferred, unselected elements
    $EX \leftarrow \{uc\} \cup$ inferUnselectedElements$(uc, d, DC, \emptyset, PLD)$
    foreach ($dm \in DC$) do
        Let SUC be the set of selected use cases in dm
        Let vp be the variation point in dm
        if ($SUC \neq \emptyset$) and (($EX \cap$ getRequiredElements(vp, PLD))$\neq \emptyset$) then
            $RES \leftarrow RES \cup \{dm\}$
        end if
        foreach ($u \in SUC$) do
            Let I be the empty set for inferred, selected elements
            $I \leftarrow \{u\} \cup$ inferSelectedElements$(u, dm, DC \cup \{d\}, \emptyset, PLD)$
            foreach ($e \in I$) do
                if ($EX \cap$ getRequiredElements(e, PLD) $\neq \emptyset$) then
                    $RES \leftarrow RES \cup$ getInvolvedDecisions$(e, dm, d, DC)$
                end if
            end foreach
        end foreach
    end foreach
end if
return RES
```

Alg. 19: inferSelectedElements

```
Input  : Traversed element (Variation Point or Use Case), $elem$,
         Set of decisions, $DC$,
         Decision, $d$,
         Set of previously traversed elements, $PR$,
         Product Line use case diagram, $PLD$
Output: Set of inferred, selected elements
Let $RES$ be the empty set for inferred, selected elements
if $(elem \notin PR)$ then
    $PR \leftarrow PR \cup \{elem\}$
    if $(elem$ is a Variation Point$)$ then
        Let $SU$ be the set of selected use cases of $elem$ in $\{d\} \cup DC$
        if $((elem$ is mandatory$)$ or $(SU \neq \emptyset))$ then
            $RES \leftarrow \{elem\}$
            Let $Y$ be the set of variant use cases of $elem$ with a mandatory variability relation
            Let $T$ be the set of variant use cases of $elem$ selected in $DC \cup \{d\}$
            $U \leftarrow Y \cup T$
            foreach $(uc \in U)$ do
                $RES \leftarrow RES \cup$ inferSelectedElements($uc$, $DC$, $d$, $PR$, $PLD$)
            end foreach
        end if
    else
        if $(elem$ is a Use Case$)$ then
            if $((elem$ is mandatory$)$ or $(elem$ is selected in $\{d\} \cup DC))$ then
                $RES \leftarrow \{elem\}$
                Let $V$ be the set of variation points included by $elem$ in $PLD$
                foreach $(vp \in V)$ do
                    $RES \leftarrow RES \cup$ inferSelectedElements($vp$, $DC$, $d$, $PR$, $PLD$)
                end foreach
            end if
        end if
    end if
end if
return $RES$
```

Alg. 20: inferUnselectedElements

```
Input  : Traversed element (Variation Point or Use Case), elem,
         Set of decisions, DC,
         Decision, d,
         Set of the previously traversed elements, PR,
         Product Line use case diagram, PLD
Output: Set of inferred, unselected elements

Let RES be the empty set for inferred, unselected elements
if ($elem \notin PR$) then
    $PR \leftarrow PR \cup \{elem\}$
    if (elem is a Variation Point) then
        Let SU be the set of variant use cases of elem selected in $\{d\} \cup DC$
        if (($SU = \emptyset$) and (there is no more decisions to be made for use cases of elem)) then
            $RES \leftarrow \{elem\}$
            Let U be the set of the use cases of elem that are unselected in $DC \cup \{d\}$
            foreach ($uc \in U$) do
                $RES \leftarrow RES \cup$ inferUnselectedElements(uc, DC, d, PR, PLD)
            end foreach
        end if
    else
        if (elem is a Use Case) then
            if ((elem is unselected in $\{d\} \cup DC$) and (there is no more decisions to be made for elem)) then
                $RES \leftarrow \{elem\}$
                Let V be the set of variation points included by elem in PLD
                foreach ($vp \in V$) do
                    $RES \leftarrow RES \cup$ inferUnselectedElements(vp, DC, d, PR, PLD)
                end foreach
            end if
        end if
    end if
end if
return RES
```

Appendix B

Algorithm for Identification of Subsequent Decision Restrictions

In this appendix, we provide the algorithm for the identification of subsequent decision restrictions. The algorithm is presented in six functions. These functions (Algs. 2-7) are called in the algorithm *inferDecisionRestrictions* (Alg. 21) to infer restrictions on subsequent decisions. The restrictions are imposed by the *require* and *conflict* relations specified in the product line use case diagram.

- inferConflictingUC (Alg. 23) and inferConflictingVP (Alg. 25) infer decision restrictions with regards to the *conflict* relation for selected variant use cases and variation points in a decision *d*, respectively;
- inferRequiringUC (Alg. 27) and inferRequiringVP (Alg. 26) infer decision restrictions with regards to the *requires* relation for unselected variant use cases and variation points in a decision *d*, respectively;
- inferRequiredByUC (Alg. 22) and inferRequiredByVP (Alg. 24) infer decision restrictions with regards to the *require* relation for selected variant use cases and variation points in a decision *d*, respectively.

We also provide in this appendix the algorithm for checking decisions restrictions. Alg. 28 identifies (1) contradicting restrictions regarding the selection and unselection of the same variant use cases in a given variation point, and (2) restrictions violating cardinality constraints in variation points.

Alg. 21: inferDecisionRestrictions

```
Input  : Set of diagram decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)
$DC \leftarrow D$
Let $IR$ be the empty set for inferred decision restrictions
while ($DC \neq \emptyset$) do
    Let $d$ be a decision in $DC$
    Let $vp$ be the variation point in decision $d$
    Let $SUC$ be the set of selected variant use cases in decision $d$
    Let $NSUC$ be the set of unselected variant use cases in decision $d$
    Let $SE$ be the set of variant use cases automatically selected when the variant use cases in
    $SUC$ are selected in decision $d$
    Let $NSE$ be the set of variant use cases automatically unselected when the variant use cases
    in $NSUC$ are unselected in decision $d$
    foreach ($u \in (SUC \cup SE)$) do
        $IR \leftarrow IR \cup$ inferRequiredByUC ($u, D, PLD$)
        $IR \leftarrow IR \cup$ inferConflictingUC ($u, D, PLD$)
    end foreach
    foreach ($u \in (NSUC \cup NSE)$) do
        $IR \leftarrow IR \cup$ inferRequiringUC ($u, D, PLD$)
    end foreach
    if ($SUC \neq \emptyset$) then
        $IR \leftarrow IR \cup$ inferRequiredByVP ($SUC, vp, D, PLD$)
        $IR \leftarrow IR \cup$ inferConflictingVP ($SUC, vp, D, PLD$)
    else
        $IR \leftarrow IR \cup$ inferRequiringVP ($NSUC, vp, D, PLD$)
    end if
    $DC \leftarrow DC \setminus \{d\}$
end while
return $IR$
```

Alg. 22: inferRequiredByUC

```
Input  : Use case ($u$), Set of decisions ($D$)
         PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)

Let $IR$ be the empty set for inferred decision restrictions
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$ and $b$ is a boolean variable
Let $R$ be the empty set of required elements
$R \leftarrow$ getRequiredElements($u$, $PLD$)
if ($R \neq \emptyset$) then
    foreach ($p \in PLD$) do
        if (p is a variation point ) then
            if ((there is a subsequent decision to be made for p) and (none of the variant use cases in p has been selected in prior decisions in D) and ($p \in R$)) then
                $IR \leftarrow IR \cup \{(null, p, true)\}$
                Let $UC$ be the set of variant use cases automatically selected in $p$ when there is a decision made for $p$
                $IR \leftarrow IR \cup$ inferConflictingVP($UC$, $p$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiredByVP($UC$, $p$, $D$, $PLD$)
                foreach ($vc \in UC$) do
                    $IR \leftarrow IR \cup$ inferConflictingUC($vc$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiredByUC($vc$, $D$, $PLD$)
                    Let $AUS$ be the set of variant use cases that are automatically selected when $vc$ is selected
                    foreach ($y \in AUS$) do
                        Let $x$ be the variation point of $y$
                        $IR \leftarrow IR \cup$ inferConflictingUC($y$, $D$, $PLD$)
                        $IR \leftarrow IR \cup$ inferRequiredByUC($y$, $D$, $PLD$)
                        $IR \leftarrow IR \cup$ inferConflictingVP($\{y\}$, $x$, $D$, $PLD$)
                        $IR \leftarrow IR \cup$ inferRequiredByVP($\{y\}$, $x$, $D$, $PLD$)
                    end foreach
                end foreach
            end if
        end if
        if (p is a variant use case ) then
            if ((there is a subsequent decision to be made for p) and (p has not been selected in prior decisions in D) and ($p \in R$)) then
                Let $e$ be the variation point of $p$
                $IR \leftarrow IR \cup \{(p, e, true)\}$
                $IR \leftarrow IR \cup$ inferConflictingUC($p$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiredByUC($p$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferConflictingVP($\{p\}$, $e$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiredByVP($\{p\}$, $e$, $D$, $PLD$)
                Let $AUS$ be the set of variant use cases that are automatically selected when the variant use case $p$ is selected
                foreach ($c \in AUS$) do
                    Let $vp$ be the variation point of c
                    $IR \leftarrow IR \cup$ inferConflictingUC($c$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiredByUC($c$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferConflictingVP($\{c\}$, $vp$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiredByVP($\{c\}$, $vp$, $D$, $PLD$)
                end foreach
            end if
        end if
    end foreach
end if
return $IR$
```

Alg. 23: inferConflictingUC

```
Input  : Use case ($u$), Set of decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$ and $b$ is a boolean variable
Let $IR$ be the empty set for inferred decision restrictions
Let $CVP$ be the set of variation points conflicting with $u$
Let $CUC$ be the set of variant use cases conflicting with $u$
foreach ($c \in CUC$) do
    if ((there is a subsequent decision to be made for c) and (c has not been selected in prior decisions in D)) then
        Let $vp$ be the variation point of $c$
        $IR \leftarrow IR \cup \{(c, vp, false)\}$
        $IR \leftarrow IR \cup$ inferRequiringUC($c$, $D$, $PLD$)
        $IR \leftarrow IR \cup$ inferRequiringVP($\{c\}$, $vp$, $D$, $PLD$)
        Let $AUC$ be the set of variant use cases automatically unselected when $c$ is unselected
        foreach ($a \in AUC$) do
            Let $p$ be the variation point of $a$
            $IR \leftarrow IR \cup \{(a, p, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($a$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{a\}$, $p$, $D$, $PLD$)
        end foreach
    end if
end foreach
foreach ($p \in CVP$) do
    if ((there is a subsequent decision to be made for p) and (none of the variant use cases in p has been selected in prior decisions in D)) then
        Let $UC$ be the set of variant use cases in $p$
        $IR \leftarrow IR \cup \{(null, p, false)\}$
        $IR \leftarrow IR \cup$ inferRequiringVP($UC$, $p$, $D$, $PLD$)
        foreach ($vc \in UC$) do
            $IR \leftarrow IR \cup$ inferRequiringUC($vc$, $D$, $PLD$)
        end foreach
        Let $AU$ be the set of variant use cases automatically unselected when the variant use cases in $p$ are unselected
        foreach ($vuc \in AU$) do
            Let $vp$ be the variation point of $vuc$
            $IR \leftarrow IR \cup \{(vuc, vp, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($vuc$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{vuc\}$, $vp$, $D$, $PLD$)
        end foreach
    end if
end foreach
return $IR$
```

Alg. 24: inferRequiredByVP

```
Input  : Set of selected use case cases ($SUC$)
         Variation Point ($vp$), Set of decisions ($D$)
         PL use case diagram ($PLD$)
Output: Set of inferred decisions restrictions ($IR$)
Let IR be the empty set for inferred decision restrictions
Let a triple $(uc, vpo, b)$ denote a decision restriction where uc is a variant use case, vpo is the variation point of uc and b is a boolean variable
Let R be the empty set for required elements
$R \leftarrow$ getRequiredElements($vp$, $PLD$)
if ($R \neq \emptyset$) then
    foreach ($p \in PLD$) do
        if (p is a variation point ) then
            if ((there is a subsequent decision to be made for p) and (none of the variant use cases in p has been selected in prior decisions in D)
            and ($p \in R$)) then
                $IR \leftarrow IR \cup \{(null, p, true)\}$
                Let UC be the set of variant use cases automatically selected in p when there is a decision made for p
                $IR \leftarrow IR \cup$ inferConflictingVP($UC$, $p$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiredByVP($UC$, $p$, $D$, $PLD$)
                foreach ($vc \in UC$) do
                    $IR \leftarrow IR \cup$ inferConflictingUC($vc$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiredByUC($vc$, $D$, $PLD$)
                    Let AUS be the set of variant use cases that are automatically selected when vc is selected
                    foreach ($y \in AUS$) do
                        Let x be the variation point of y
                        $IR \leftarrow IR \cup$ inferConflictingUC($y$, $D$, $PLD$)
                        $IR \leftarrow IR \cup$ inferRequiredByUC($y$, $D$, $PLD$)
                        $IR \leftarrow IR \cup$ inferConflictingVP($\{y\}$, $x$, $D$, $PLD$)
                        $IR \leftarrow IR \cup$ inferRequiredByVP($\{y\}$, $x$, $D$, $PLD$)
                    end foreach
                end foreach
            end if
        end if
        if (p is a variant use case ) then
            if ((there is a subsequent decision to be made for p) and (p has not been selected in prior decisions in D) and ($p \in R$)) then
                Let e be the variation point of p
                $IR \leftarrow IR \cup \{(p, e, true)\}$
                $IR \leftarrow IR \cup$ inferConflictingUC($p$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiredByUC($p$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferConflictingVP($\{p\}$, $e$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiredByVP($\{p\}$, $e$, $D$, $PLD$)
                Let AUS be the set of variant use cases that are automatically selected when the variant use case p is selected
                foreach ($c \in AUS$) do
                    Let vp be the variation point of c
                    $IR \leftarrow IR \cup$ inferConflictingUC($c$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiredByUC($c$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferConflictingVP($\{c\}$, $vp$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiredByVP($\{c\}$, $vp$, $D$, $PLD$)
                end foreach
            end if
        end if
    end foreach
end if
return IR
```

Alg. 25: inferConflictingVP

```
Input  : Set of selected use case cases ($SUC$), Variation Point ($vp$), Set of decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decisions restrictions ($IR$)
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$ and $b$ is a boolean variable
Let $IR$ be the empty set for inferred decision restrictions
Let $CVP$ be the set of variation points conflicting with $vp$
Let $CUC$ be the set of variant use cases conflicting with $vp$
foreach ($p \in CVP$) do
    if ((there is a subsequent decision to be made for p) and (none of the variant use cases in p has been selected in prior decisions in D)) then
        $IR \leftarrow IR \cup \{(null, p, false)\}$
        Let $UC$ be the set of variant use cases in $p$
        $IR \leftarrow IR \cup$ inferRequiringVP($UC$, $p$, $D$, $PLD$)
        foreach ($vc \in UC$) do
            $IR \leftarrow IR \cup$ inferRequiringUC($vc$, $D$, $PLD$)
        end foreach
        Let $AUC$ be the set of variant use cases that are automatically unselected when the variant use cases in $p$ are unselected
        foreach ($vuc \in AUC$) do
            Let $q$ be the variation point of $vuc$
            $IR \leftarrow IR \cup \{(vuc, q, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($vuc$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{vuc\}$, $q$, $D$, $PLD$)
        end foreach
    end if
end foreach
foreach ($c \in CUC$) do
    Let $x$ be the variation point of $c$
    if ((there is a subsequent decision to be made for c) and (c has not been selected in prior decisions in D)) then
        $IR \leftarrow IR \cup \{(c, x, false)\}$
        $IR \leftarrow IR \cup$ inferRequiringUC($c$, $D$, $PLD$)
        $IR \leftarrow IR \cup$ inferRequiringVP($\{c\}$, $x$, $D$, $PLD$)
        Let $AUC$ be the set of variant use cases that are automatically unselected when $c$ is unselected
        foreach ($a \in AUC$) do
            Let $y$ be the variation point of $a$
            $IR \leftarrow IR \cup \{(a, y, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($a$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{a\}$, $y$, $D$, $PLD$)
        end foreach
    end if
end foreach
return $IR$
```

Alg. 26: inferRequiringVP

```
Input  : Set of variant use cases ($NSUC$), Variation point ($vp$),
         Set of Decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decisions restrictions ($IR$)
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$ and $b$ is a boolean variable
Let $IR$ be the empty set for inferred decision restrictions
Let $R$ be the empty set of required elements
Let $U$ be the set of variant use cases in $vp$
if ((NSUC = U) or (the variant use cases in ($U \setminus NSUC$) are already restricted to be unselected)) then
    foreach ($e \in PLD$) do
        $R \leftarrow$ getRequiredElements($e$, $PLD$)
        if (e is a variation point ) then
            if ((there is a subsequent decision to be made for e) and (none of the variant use cases in e has been selected in prior decisions in D)
            and ($vp \in R$)) then
                $IR \leftarrow IR \cup \{(null, e, false)\}$
                Let $UC$ be the set of variant use cases in $e$
                $IR \leftarrow IR \cup$ inferRequiringVP($UC$, $e$, $D$, $PLD$)
                foreach ($vc \in UC$) do
                    $IR \leftarrow IR \cup$ inferRequiringUC($vc$, $D$, $PLD$)
                end foreach
                Let $AU$ be the set of variant use cases that are automatically unselected when the variant use cases in $e$ are unselected
                foreach ($vuc \in AU$) do
                    Let $v$ be the variation point of $vuc$
                    $IR \leftarrow IR \cup \{(vuc, v, false)\}$
                    $IR \leftarrow IR \cup$ inferRequiringUC($vuc$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiringVP($\{vuc\}$, $v$, $D$, $PLD$)
                end foreach
            end if
        end if
        if (e is a variant use case ) then
            if ((there is a subsequent decision to be made for e) and (e has not been selected in prior decisions in D) and ($vp \in R$)) then
                Let $p$ be the variation point of $e$;
                $IR \leftarrow IR \cup \{(e, p, false)\}$
                $IR \leftarrow IR \cup$ inferRequiringUC($e$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiringVP($\{e\}$, $p$, $D$, $PLD$)
                Let $AUC$ be the set of variant use cases automatically unselected when $p$ is unselected
                foreach ($a \in AUC$) do
                    Let $y$ be the variation point of $a$
                    $IR \leftarrow IR \cup \{(a, y, false)\}$
                    $IR \leftarrow IR \cup$ inferRequiringUC($a$, $D$, $PLD$)
                    $IR \leftarrow IR \cup$ inferRequiringVP($\{a\}$, $y$, $D$, $PLD$)
                end foreach
            end if
        end if
    end foreach
end if
return $IR$
```

Alg. 27: inferRequiringUC

```
Input  : Use Case ($u$), Set of decisions ($D$), PL use case diagram ($PLD$)
Output: Set of inferred decision restrictions ($IR$)
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$ and $b$ is a boolean variable
Let $IR$ be the empty set for inferred decision restrictions
Let $R$ be the empty set of required elements
foreach $(e \in PLD)$ do
    $R \leftarrow$ getRequiredElements($e$, $PLD$)
    if (e is a variation point) then
        if ((there is a subsequent decision to be made for e) and (none of the variant use cases in e has been selected in prior decisions in D) and $(u \in R)$) then
            $IR \leftarrow IR \cup \{(null, e, false)\}$
            Let $UC$ be the set of variant use cases in $e$
            $IR \leftarrow IR \cup$ inferRequiringVP($UC$, $e$, $D$, $PLD$)
            foreach $(vc \in UC)$ do
                $IR \leftarrow IR \cup$ inferRequiringUC($vc$, $D$, $PLD$)
            end foreach
            Let $AU$ be the set of variant use cases automatically unselected when the variant use cases in $e$ are unselected
            foreach $(vuc \in AU)$ do
                Let $vp$ be the variation point of $vuc$
                $IR \leftarrow IR \cup \{(vuc, vp, false)\}$
                $IR \leftarrow IR \cup$ inferRequiringUC($vuc$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiringVP($\{vuc\}$, $vp$, $D$, $PLD$)
            end foreach
        end if
    end if
    if (e is a variant use case) then
        if ((there is a subsequent decision to be made for e) and (e has not been selected in prior decisions in D) and $(u \in R)$) then
            Let $p$ be the variation point of $e$
            $IR \leftarrow IR \cup \{(e, p, false)\}$
            $IR \leftarrow IR \cup$ inferRequiringUC($e$, $D$, $PLD$)
            $IR \leftarrow IR \cup$ inferRequiringVP($\{e\}$, $p$, $D$, $PLD$)
            Let $AUC$ be the set of variant use cases automatically unselected when $p$ is unselected
            foreach $(a \in AUC)$ do
                Let $y$ be the variation point of $a$
                $IR \leftarrow IR \cup \{(a, y, false)\}$
                $IR \leftarrow IR \cup$ inferRequiringUC($a$, $D$, $PLD$)
                $IR \leftarrow IR \cup$ inferRequiringVP($\{a\}$, $y$, $D$, $PLD$)
            end foreach
        end if
    end if
end foreach
return $IR$
```

Alg. 28: checkDecisionRestrictions

```
Input  : Set of decision restrictions ($R$), PL use case diagram ($PLD$)
Output: Set of sets of contradicting decisions ($CR$)
Let a triple $(uc, vpo, b)$ denote a decision restriction where $uc$ is a variant use case, $vpo$ is the variation point of $uc$, and $b$ is a boolean variable
Let $CR$ be the empty set for sets of contradicting restrictions
Let $VP$ be the set of variation points in $PLD$
foreach ($p \in VP$) do
    Let $DR$ be the set of decisions restrictions in $R$ for $p$
    $IR \leftarrow DR$
    foreach ($r \in DR$) do
        $IR \leftarrow IR \setminus \{r\}$
        foreach ($e \in IR$) do
            if ($e.uc = r.uc$ and $e.b \neq r.b$) then
                $CR \leftarrow CR \cup \{\{e, r\}\}$
            end if
            if ($r.uc = null$) then
                if ($r.b = false$) then
                    if ($e.b = true$) then
                        $CR \leftarrow CR \cup \{\{e, r\}\}$
                    end if
                end if
            end if
        end foreach
        if ($r.uc = null$ and $r.b = true$ ) then
            $CR \leftarrow CR \cup$ checkSeveralRestrictions($p$, $DR$, $PLD$)
        end if
    end foreach
    $CR \leftarrow CR \cup$ checkCardinality($p$, $DR$, $PLD$)
end foreach
return $CR$
```